Efficient Validation Input Generation in RTL by Hybridized Source Code Analysis

Lingyi Liu and Shobha Vasudevan
Department of ECE, University of Illinois at Urbana-Champaign, Urbana, IL, USA
{liu187, shobhav}@illinois.edu

***Abstract*—We present HYBRO, an automatic methodology to generate high coverage input vectors for Register Transfer Level (RTL) designs based on branch-coverage directed approach. HYBRO uses dynamic simulation data and static analysis of RTL control flow graphs (CFGs). A concrete simulation is applied over a fixed number of cycles. Instrumented code records the branches covered. The corresponding symbolic trace is extracted from the CFG with an RTL symbolic execution engine. A guard in the symbolic expression is mutated. If the mutated guard has dependent branches that have not already been covered, it is mutated and passed to an SMT solver. A satisfiable assignment generates a valid input vector. We implement the Verilog RTL symbolic execution engine and show that the notion of branch-coverage directed exploration can avoid path explosion caused by previous path-based approach to input vector generation and achieve full branch and more than 90% functional(assertion) coverage quickly on ITC99 benchmark and several Openrisc designs. We also describe two types of optimizations a) dynamic UD chain slicing b)local conflict resolution to speed up HYBRO by 1.6-12 times on different benchmarks.**

I. Introduction

The validation phase of a Register Transfer Level (RTL) design is widely accepted as responsible for perpetual bottlenecks in the design cycle. Application of known stimulus, or directed testing helps capture expected behavior of the design. Although developing directed tests is an arduous task involving many man-months, the directed test suites usually converge at an acceptable point. However, in the case of random stimuli, this confidence is far from being achieved. Even in state-of-the-art industrial environments with many dedicated validation resources, the design is considered as stable after the application of a large number ($>$1 trillion) of random patterns. Since this metric is devoid of any information regarding design behavior coverage, it is very unsatisfactory.

In this work, we present HYBRO (HYbrid analysis and BRanch Coverage Optimizations), a methodology to generate input vectors in RTL automatically and with extremely high coverage. In HYBRO, analysis of RTL is done using the Control Flow Graph (CFG) structure of the Hardware Description Language (HDL) source code. This means that the RTL description is analyzed by considering it as a software program, using program analysis techniques. HYBRO uses static as well as dynamic program analysis techniques for test generation in RTL. In [13], STAR, a similar technique that combines static and dynamic analysis for input vector generation was introduced. STAR generates input vector patterns for all control paths of an RTL design. However, STAR gives rise to the $path\ explosion$ challenge. Hardware RTL models the sequential design as always/process block in verilog/VHDL. In semantic, this always/process block is an endless loop. The number of paths will be increased exponentially as the number of unrolled cycles increases.

HYBRO circumvents the path explosion problem faced by STAR by considering branch coverage as the metric for guiding the path exploration. The algorithm functions as follows. A concrete simulation/execution is applied to the RTL design for a given number of cycles. The RTL design source code is instrumented with source code that gets activated if a particular branch has been taken by the concrete simulation. At the end of a concrete execution, the RTL *symbolic execution* is activated to extract the symbolic expressions along the concrete path. These contain executed $guard$(conditional expression of branch) that evaluates to true or false. In order to systematically explore the design, the extracted symbolic expressions are placed onto a constraint stack and one of the guards is *mutated* or inverted. The mutated symbolic expression is passed as a constraint through a Satisfiability Modulo Theory [2](SMT) solver to get a satisfiable assignment corresponding to the next input vector. The STAR terminates when all the conditional expressions in the RTL or *guards* in the constraint stack have been exhausted.

In contrast, HYBRO uses a coverage driven approach to mutate a guard and give a symbolic expression to the SMT solver. In HYBRO, *the instrumented code is also used to record branch coverage* in the RTL CFG. At the stage when a guard is picked for mutation, if all the branches in the CFG that depend on the mutated guard have already been covered, a different guard is picked from the symbolic expression. HYBRO tries to use the guidance provided by branch coverage to stimulate all reachable branches in the CFG, it does not guarantee complete coverage. It can be viewed as a best effort process that practically guide the generation process to uncovered regions and produces excellent coverage. This makes the analysis much more efficient as compared to STAR.

Additionally, in this work, we also present two optimizations that increase the efficiency of HYBRO. Both of them draw on the static analysis and dynamic analysis technology. The first optimization is dynamic UD(Use-Definition) chain slicing. The UD chain is a data structure consisting of a use(U) of a variable and all definitions(D) of the variable that can reach the use without any other intervening definitions. This approach removes redundant constraint in the path constraints. The second optimization involves resolving local conflicts when making guard mutation. The successful detection of conflict

978-3-9810801-7-9/DATE11/

can reduces calls to the SMT solver [2].

It must be noted that we perform static analysis and symbolic execution at the RT-level. This means that we use word level operators abstractions. Our entire analysis is at this higher level, as opposed to traditional RT-Level test generation techniques that use gate level Boolean reasoning. This is the reason to use an SMT solver instead of a Boolean Satisfiability (SAT) solver. RTL constructs like the guard at every branch statement provide guidance about higher level design intent. Therefore, the patterns generated by HYBRO provide high functional coverage of the design. In addition, since reasoning at higher levels of abstraction provides a more macroscopic view of the design and is more time efficient, HYBRO is capable of generating interesting input patterns quickly.

We also introduce symbolic execution for RTL, a concept that was previously defined only in software. This is distinct from RTL symbolic simulation [11] [12], which has been used in hardware before. Symbolic execution follows a single execution path in the design, as opposed to symbolically simulating all paths pertaining to each process in the design. This results in a more efficient engine than a symbolic simulator. Also, since it follows the dynamic, concrete execution, it only considers feasible paths, a luxury that a static engine like a symbolic simulator would not have.

Our contributions in this work are as follows:

- We present HYBRO, an automatic, efficient technique to generate high coverage input patterns in RTL by using a hybrid approach that combines dynamic and static analysis of the RTL source code.
- HYBRO uses a branch-coverage directed approach to heuristically guide the region of exploration for the input vector generation. We track branch coverage in HYBRO using a code instrumentation methodology.
- We introduce symbolic execution for RTL. We have implemented a symbolic execution engine in Verilog RTL for the extraction of symolic expressions in HYBRO.
- We present two optimizations to the base HYBRO algorithm that increase the efficiency of the technique-dynamic UD chain slicing and local conflict resolution.
- The static analysis, symbolic execution and SMT-based constraint solving in HYBRO are performed completely at the RT-level. HYBRO is able to provide functionally relevant patterns due to a region-wise guard-based coverage strategy.

An outline of the paper is as follows. In section 2 and 3, we present related work and background information. In section 4, we describe the implementation of RTL symbolic execution engine. In section 5, we detail our HYBRO approach step by step and also include the optimizations. Section 6 demonstrates the experimental evidence of the merit of our approach.

II. Related work

We discuss work that is related to different aspects of HYBRID. To the best of our knowledge, the integration of these aspects in the design validation space has not been presented before. In software testing, there is extensive research on the idea combining concrete execution and symbolic execution [14] [10]. In hardware, hybrid techniques that combine dynamic (simulation) with static analysis at the gate level have been used for formal verification [8] and commercial tools like Zero-In.

Static analysis of RTL has been used for verification [4] [15] and manufacturing level testing [17]. Symbolic simulation technology was initially used at gate-level for formal verification [3] [11]. Recently, some researchers tried to develop the RTL symbolic simulator [12]. Different from these work, HYBRO's symbolic execution engine considers one feasible path each time instead of the entire design and is more scalable to large design. There is also some prior work leveraging the static structure of RTL to speed up model checking [6].

Coverage-guided stimulus generation [7] and development of effective coverage metrics [18] has been looked at in depth before. Test generation in RTL targeting at stuck-at manufacturing faults [9] has been explored to reduce test generation time.

III. Background concepts and definitions

We treat the RTL source code as a "program" as in [4] [15]. We analyze the CFG of the RTL design. A *simple path* in the RTL CFG is a path which is executed in a single cycle. A *sequential path* refers to a path that is executed across multiple time cycles. In order to account for the sequential behavior of the RTL, the RTL CFG is *sequentially unrolled*. This means that the CFG is replicated many times, with the variables in each unroll being annotated by the corresponding relative time cycle.

In the CFG, a conditional node i is *control dependent* on conditional node j if the outcome of evaluating j determines if i should execute or not. j is said to dominate i. A *control dependency graph* is a data structure that maintains the control dependencies within a single time cycle. As shown in Figure 1, the broken line indicates the control dependency.

IV. Verilog RTL Symbolic execution engine implementation

Symbolic execution [5] refers to the execution of a single path with symbolic inputs. Symbolic execution of a path generates *symbolic expressions* that are a logical conjunction of the guards and assignments to the variables used in guards along that path. Symbolic execution of a sequential path generates expressions such that every variable along the path is annotated by the time cycle to which it belongs. A *constraint* in our context is a symbolic expression translated in a form that is acceptable by an SMT constraint solver. The symbolic execution described here considers only the synthesizable subset of Verilog, which make the design of execution engine easier and the generated symbolic expression compatible with SMT solver. The entire execution engine is working on the CFG and expression tree structure of each statement.

A. CFG and Expression Tree Structure

Figure 1 (a)shows a Verilog RTL example design with instrumented code. For each single statement or conditional

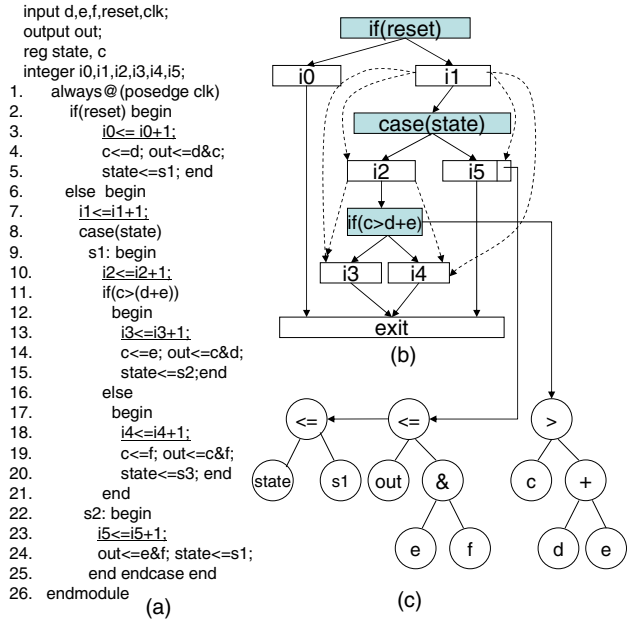

Fig. 1. An RTL example with instrumented code and its corresponding CFG and expression tree

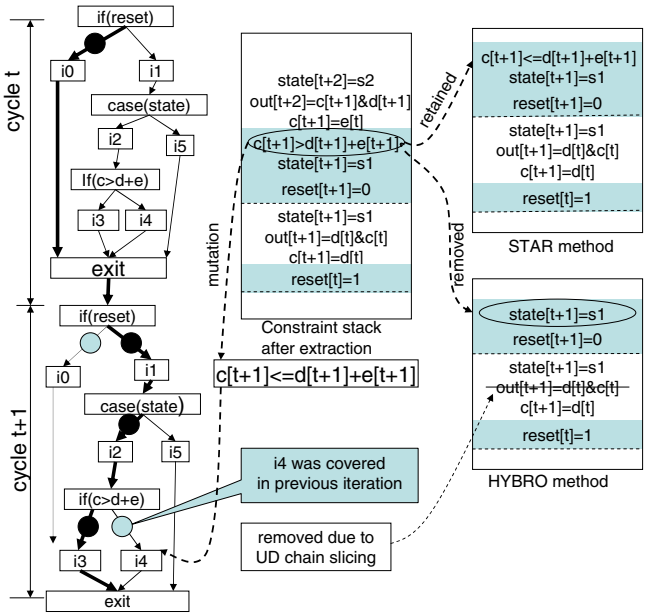

Fig. 2. Branch coverage guided search approach in HYBRO. A comparison to the STAR algorithm is shown.

expression in the design, the expression tree structure exactly records the corresponding assignment or expressions for later constraint generation and is linked to corresponding CFG node. As shown in Figure 1, the conditional expression in line *11* is represented in the expression tree linked to the branch node. All the nonblocking statements in line *24* are represented in the expression tree structures that are linked to the corresponding CFG node *i5*. The expression tree can also be used to build the use-define chain for the design since it is easy to deduce the used variable and defined variable from expression tree. For example, in a non assignment expression, all leaf node variables are the used variables in the expression.

In addition, shown in diamond in CFG in Figure 1 (b), all the instrumented branch variables keep track of the concrete simulation path at each cycle by sustaining an array which is indexed by cycle number. At the end of each cycle, the instrumented branch variables are compared with their value in the last cycle. The updated variable means the corresponding branch is taken, which is recorded in corresponding element in the array. When the concrete simulation is done, the symbolic execution can exactly follow the executed concrete path.

B. Path Constraint Generation

The symbolic execution engine walks the CFG in the design one by one at every cycle. In each CFG, it only follows the concrete simulation path. At each branch node in each cycle, the engine decides the taken path by looking up the corresponding element in the array. At each node in the path, the corresponding expression tree is traversed and output as symbolic expression. Shown in Figure 1 (b), the engine arrives at the CFG branch node *if(c>d+e)* in cycle *i* and traversing of linked expression tree can generate the following constraint: *c[i]>d[i]+e[i]*.

For nonblocking assignments to register variables in the path, the assigned register variable will take effect in next cycle. Therefore, the cycle index should be set to the next cycle number. Taking the assignment *out<=e&f* in node *i5* for example, the generated constraint will be *out[i+1]==e[i]&f[i]*. Finally, the conjunction of all generated symbolic expressions forms the corresponding path constraints.

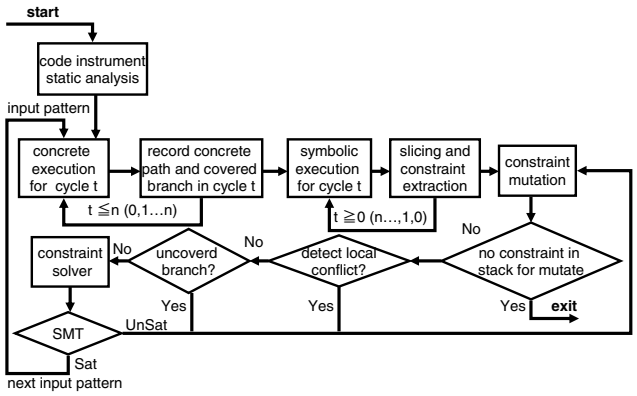

Fig. 3. The HYBRO algorithm flow

V. HYBRO: Branch Coverage Guided Input Generation Approach

Figure 3 shows the algorithmic flow of HYBRO. We describe every phase of the algorithm briefly below.

A. Code instrumentation and static analysis

In this phase, the CFG of the given RTL design is analyzed to obtain the UD chain as well as the control dependency graph. Additionally, the RTL design is instrumented with source code that is meant to trace a concrete execution path. In Figure 1 (a), the instrumented code has been underlined. The values of $i0$ will change if $reset$ evaluates to 1, and the value of $i1$ will change if $reset$ evaluates to 0. A change in the value of either $i0$ or $i1$ indicates which branch was executed by the concrete simulation. In Figure 1 (b), a CFG

is shown where the control dependency is marked by dotted lines. The instrumented code also forms a part of the CFG. It serves as a communication channel between the concrete and symbolic executions of a path. The instrumentation is automatically done by a verilog parser.

B. Concrete execution for multiple cycles and recording branch coverage

A concrete input pattern is given as stimulus. For the first time, this stimulus is generated at random. For every subsequent iteration of the algorithm, the concrete input patterns are generated automatically. The concrete stimulus is applied for a predetermined number of cycles. Every time a branch executes in the concrete input simulation, the edges corresponding to the instrumented code in the CFG are marked as covered. In the concrete execution shown in Figure 2, the corresponding concrete pattern is $reset[t] = 1$, $d[t] = 1$, $e[t] = 1$, $f[t] = 0$; $reset[t+1] = 0$, $d[t+1] = 0$, $e[t+1] = 0$, $f[t+1] = 1$. The bold edges correspond to the concrete execution path in the CFG. In the first cycle $reset = 1$ is applied, so the edge leading to $i0$ is marked as covered. This is shown by the large dot on the arrows. In the second cycle $reset = 0$ is applied, and the branches leading to $i1$, $i2$ and $i3$ are marked as covered. The branch leading to node $i4$ is marked as covered from a previous iteration of the algorithm.

C. Symbolic execution

In this step, the concrete path identified in the control flow graph is symbolically executed. For the concrete path executed over multiple cycles, the corresponding symbolic execution will also involve variables across multiple time cycles. For the example concrete path, the symbolic execution will yield the following expression. In cycle t: $reset[t] = 1 \bigwedge c[t+1] = d[t] \bigwedge out[t+1] = d[t] \& c[t] \bigwedge state[t+1] = s1$. In cycle $t+1$: $reset[t+1] = 0 \bigwedge state[t+1] = s1 \bigwedge c[t+1] > d[t+1] + e[t+1] \bigwedge c[t+2] = e[t+1] \bigwedge out[t+2] = c[t+1] \& d[t+1] \bigwedge state[t+2] = s2$. The $state[t+1] = s1$ appearing in cycle t corresponds to the non-blocking assignment in line 5 in Figure 1 (a). Its appearing in cycle $t+1$ corresponds to the guard in line 9 in Figure 1 (a).

D. Dynamic UD chain slicing and constraint extraction

The regular constraint extraction mechanism would be to simply reuse the symbolic expression. However, we introduce an optimization strategy here that makes use of the UD chain. We traverse the CFG from the last time cycle backwards to the first time cycle in current dynamic execution to apply this optimization. For every variable in every guard in a cycle we refer the UD chain to see where it was defined. Among all possible definitions for a used guard variable found by static analysis, we only consider the one that has been executed by the concrete input vector. We mark all the definitions transitively from the last cycle to the first cycle. At the end of this analysis, if a definition has been marked, it must be required by a guard in a subsequent frame. Otherwise, it is discarded from the constraint.

The constraint is extracted into the *constraint stack* such that every element of the stack corresponds to a term that is a conjunct in the symbolic expression. For example, in Figure 2 the constraint is pushed into the constraint stack such that each element is annotated with the cycle number and the lowest cycle number is at the bottom of the stack. Since UD chain slicing analyzes the CFG from last cycle to first cycle, the constraint stack elements need to be popped and then re-pushed into the stack. The UD chain slicing optimization is intended to make the size of the constraint smaller. In Figure 1 (a), there are four definitions in the constraint in cycle t for the used variable c in line 11 in cycle $t+1$. They are $c <= d$ in line 4, $c <= e$ in line 14, $c <= f$ in line 19 and an implicit definition $c <= c$ in $i5$. However, only $c <= d$ in line 4 in cycle t is extracted as a constraint since it is in the concrete path. In addition, the definition $out[t+1] = d[t] \& c[t]$ can be removed from the constraint since variable out is not used in the following cycle.

E. Constraint mutation and branch coverage

A constraint is said to be mutated if any of its guards is inverted/mutated. In this step, the guard at the top of the constraint stack is selected as a candidate for mutation. The candidate guard is mutated and then analyzed using the CFG that was marked by branch coverage as follows. If the mutated candidate guard g corresponds to a control node in the control dependency graph, and all the branches leading to the nodes that are control dependent on g have already been covered, then g is discarded from the constraint stack. If any of the nodes dependent on g can be executed by branches that have *not yet been covered*, g is retained. Intuitively, if B is control dependent on A, it means that some path in the program that goes through A can bypass B, and A is the point in which this divergence can occur. So, it suffices to look at the control node that dominates the other nodes for doing a branch overage analysis. The branch coverage analysis is performed for only one cycle at a time. The cycle that is considered for a guard corresponds to the annotated in the guard variable. So, in the example, only the $(t+1)^{th}$ unroll is analyzed for control dependency and branch coverage.

In Figure 2, the guard $c[t+1] > d[t+1] + e[t+1]$ is the mutation candidate. Definitions/Assignments in the constraint are not considered mutation candidates. The guards in the constraint are shown by the shaded elements of the stack. These will be mutation candidates. The candidate guard is mutated to $c[t+1] \leq d[t+1] + e[t+1]$. The mutated guard now corresponds to the node $i4$. We first check if $i4$ dominates other uncovered conditional nodes(including itself). However, $i4$ is marked as covered in previous iteration of the algorithm. So this guard is removed from the constraint in our approach. In future cycles (t+2 and beyond) of the algorithm, $i4$ might dominate other control nodes according to the control dependency graph in Figure 1 (b). If there are control dependent nodes that are not yet covered, the current mutated candidate guard will be retained.

As shown in Figure 2, the STAR algorithm would have retained this guard, irrespective of it being along a branch that has been covered. This would result in repetitive coverage of the paths that execute control nodes that are dependent on $i4$

in all future iterations as well. Our approach manages to avoid repetitive path traversal for input pattern generation by using the notion of branch coverage.

F. Local conflict resolution

There are two kinds of local conflict when making guard mutation. First, the same guard occurs across multiple processes in the same cycle in RTL. If this guard is part of the constraint stack and gets mutated, this will result in a conflict with the same guard that is present lower in the stack. Similarly, if the previous definition(s) of a used variable in a guard are assigned a constant value, the mutation of that guard to another value will give rise to a conflict. We detect syntactically equivalent guards shared across multiple processes in a single cycle when doing static analysis. If such a shared guard is a candidate for mutation, it is directly popped out from the stack. Another local conflict occurs when all used variables in current mutated guard are assigned a constant value in the variable's definition of current path. For example, before the mutation of guard $a>b$, we first trace the definition of a and b through UD chain. If both of the definitions assign a constant value to a and b, the mutation of $a>b$ will definitely lead to a local conflict. This case often takes place for $case$ statement in the design. In the example shown in Figure 2, if the guard variable $state$ is a candidate for mutation, we can pop out the guard from the constraint stack since $state$ is assigned a constant in its definition. These local conflicts are supposed to be detected in the SMT solver phase. However, as an optimization, we detect such conflicts before we pass the constraint to the SMT solver.

G. Constraint solving and next pattern generation

The mutated constraint is passed through an SMT solver. If a satisfiable assignment is generated, this corresponds to the next concrete input pattern. If some inputs have been removed from the stack in the guard mutation phase, they will not be a part of the constraint. These might need to be randomly generated in the next concrete pattern. However, the random generation along with the existing constraints will direct the entire test generation into another *region* of the CFG. The branch coverage metric, therefore, provides the benefit of balancing the extent of coverage on different parts of the RTL. The entire algorithm will be repeated for the new region. The algorithm terminates when there is no guard in the stack that has not been mutated.

VI. Experimental evaluation

We have implemented the HYBRO algorithm and all optimization strategies with C++, which interact with VCS simulator through the direct programming interface(DPI) and Yices [2] constraint solver with its C Library Interface. All the following experiments are performed on a four Intel i5 2.67GHz processor cores machine with 16GB of memory running Linux. We present a set of experimental results on some examples of RTL model from ITC99 and OpenRISC1200 [1]. Four OR1200-x designs are instruction cache controller, data cache controller , Wishbone bus interface and exception handling logic.

Bench-mark	HYBRO				
	Cycles	Bran_Cov	Path_Cov	Assert_Cov	Runtime
b01	10	94.44%	94.44%	95%	0.07s
b06	10	94.12%	93.10%	100%	0.10s
b10	10	87.10%	72.73%	4.71%	4.56s
b10	30	96.77%	81.82%	68.58%	52.14s
b10	50	96.77%	81.82%	93.65%	180.42s
b11	10	78.26%	78.26%	43.97%	0.28s
b11	50	91.30%	91.30%	100%	326.85s
b14	15	83.50%	13.36%	100%	301.69s
or1200-0	50	93.75%	77.78%	100%	37.73s
or1200-0	100	93.75%	77.78%	100%	191.82s
or1200-1	50	96.30%	79.07%	94.12%	21.90s
or1200-1	100	96.30%	79.07%	100%	92.15s
or1200-2	10	100%	100%	100%	302.67s
or1200-3	5	91.53%	90.20%	96.67%	19.07s
or1200-3	10	96.61%	96.08%	100%	287.62s

TABLE I
THE COVERAGE, RUNNING TIME, NUMBER OF PATTERNS AND REPEATED BRANCHES REPORTED BY HYBRO

A. Structural Coverage evaluation

The first experiment in Table I shows the coverage rate for the generated test patterns using HYBRO. It can be observed that HYBRO can achieve very high structural coverage as long as the unrolled cycle number is enough. For most of these designs, all the feasible branches in the design are fully covered even if the tool does not report 100% coverage due to the infeasible pathes. For example, there maybe unreachable $default$ branch for $case$ statement in a design.

The unrolled cycle number is an important parameter to improve the coverage in HYBRO. This parameter is determined by the coverage feedback. If the coverage is not high, it means the uncovered branches are not reachable in the unrolled cycles. We can increase the unrolled cycle number. The $b10$ and $b11$ circuit demonstrate this relationship between coverage and unrolled cycle number. When the unrolled cycle number increases from 10 to 30, all the feasible branches are fully covered. However, for or1200-2 and or1200-3 designs, 10 cycles is enough to cover all branches. The running time exhibits the applicability of HYBRO for practical circuit. There is no memory bottleneck since HYBRO does not store any states of the circuit.

The only exceptionable design is $b14$ circuit. In this design, several uncovered branch conditions in the design depend on the overflow of a big counters. As a result, it becomes difficult to satisfy these branch conditions.

A very interesting benefit from HYBRO is that it can identify and report infeasible paths. This is highly valuable to the verification engineer. In addition, HYBRO can also be used to check properties on each path.

B. Functional coverage analysis

We also demonstrate that the input patterns generated by HYBRO can provide high functional coverage although the entire analysis in HYBRO is on RTL structure. The RTL code structures reflect the design specification and HYBRO is able to follow these structures to generate meaningful patterns from the perspective of interesting functionality.

We use the assertion tool GoldMine [16] to generate assertions for several designs. The generated tests are also applied

Bench-mark	Cycles	STAR			HYBRO			HYBRO Optimization detail		
		Runtime	Pattern num	Repeated Branch	Runtime	Pattern num	Repeated Branch	UD chain	Local conflict	speedup
b01	10	1.64s	1024	1249	0.07s	20	24	56.10%	69	12.5
b10	15	>3600s	-	-	10.8s	430	498	34.60%	472	8.01
b11	15	293.00s	84342	111736	1.12s	302	394	12.64%	1169	4.47
or1200-0	10	>3600s	-	-	1.72s	117	185	2.30%	42	1.68
or1200-3	10	>3600s	-	-	287.62s	1141	2640	10.67%	3933	1.62

TABLE II
COMPARISON BETWEEN HYBRO AND STAR AND HYBRO OPTIMIZATION DETAIL.

on the design to evaluate the assertion coverage through simulation. The assertions are normally used to check whether the design has correctly implemented the specification. The triggered assertion coverage reflects the quality of simulation patterns. The assertion coverage report of our generated test patterns by HYBRO is shown in the assertion coverage column of Table 1, from which we can conclude that HYBRO is able to comprehensively capture the design function.

Comparing with random test generation, HYBRO has the advantage of generating all meaningful inputs to cover all possible design behaviors. The generating process relies on the hint from RTL structure. Comparing with widely used constraint-random generation method, HYBRO doesn't have to manually build the constraints.

C. Optimization effects

The second experiment shown in table II is a comparison between HYBRO and the STAR algorithm [13]. All runtime are in seconds. The length of each generated patterns is equal to the unrolled cycle number. We set the maximum runtime to one hour. It can be observed that STAR suffers from path explosion for most of the designs. This is shown by high runtime and large number of patterns. The runtime will increase exponentially as the unrolled cycle number increases in STAR. Therefore, the unrolled cycle number in this experiment is very small when applying STAR. It is unscalable for big practical designs. For the same circuit, however, the runtime for HYBRO is very small. The runtime of HYBRO does not increase exponentially with the unrolling cycle number. We can see that the number of repeated branches is very high in STAR, whereas in HYBRO, the average number of repetitively covered branch is less than 5% of that in STAR.

The optimization detail column in table II shows the optimization effectiveness of UD chain slicing and local conflict resolution. UD chain column represents the percentage of reduced constraint numbers. Local conflict column represents the number of detected conflict when mutating constraint. The speedup column is the running time speedup of HYBRO with two optimizations over HYBRO without two optimizations. UD chain slicing reduces the number of constraints sent to SMT solver since unused variable definition will be excluded from current constraints. Local conflict resolution reduces the number of calling to SMT solver since it relies on the static analysis information to check the conflict in constraints stack instead of sending them to SMT solver. We specifically run HYBRO with the two optimizations and without the two optimizations. It can be observed that the optimizations can speed up HYBRO by 1.6-12times on various circuits.

VII. CONCLUSIONS

We have presented a scalable, efficient input vector generation strategy that provides very high structural and functional coverage. The main novelty of this technique is in the hybrid analysis between concrete simulation data and static analysis of the RTL code and the heuristical branch guided path exploration. We believe that this technique is highly powerful and can be applied to large scale contemporary designs.

REFERENCES

[1] Openrisc web page: *http://www.opencores.org*.

[2] Yices web page: *http://yices.csl.sri.com/*.

[3] R. E. Bryant. Symbolic simulation - techniques and applications. In *Proc. of DAC*, pages 517–521, 1990.

[4] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In *Correct Hardware Design and Verification Methods*, pages 298–312, 1999.

[5] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transaction on Software Engineering*, pages 215–222, 1976.

[6] F. Somenzi and D. Ward. Decomposing image computation for symbolic reachability analysis using control flow information. In *Proc. of ICCAD*, 2006.

[7] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Proc. of DAC*, pages 286–291, 2003.

[8] M. K.Ganai, P. Yalagandula, A. Aziz, A. Kuehlmann, and V. Singhal. Siva: A system for coverage-directed state space search. *Journal of Electronic Testing: Theory and Applications*, pages 11–27, 2001.

[9] I. Ghosh and M. Fujita. Automatic test pattern generation for functional register-transfer level circuits using assignment decision diagrams. *IEEE Transaction on Computer-Aided Design*, pages 402–415, 2001.

[10] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN PLDI*, 2005.

[11] R. B. Jones. *Applications of Symbolic Simulation to the Formal Verification of Microprocessors*. PhD thesis, Stanford University, 1999.

[12] A. Koelbl, J. H. Kukula, and R. Damiano. Symbolic rtl simulation. In *Proc. of DAC*, pages 47–50, 2001.

[13] L. Liu and S. Vasudevan. Star: Generating input vectors for design validation by static analysis of rtl. In *IEEE HLDVT Workshop*, 2009.

[14] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proc. of the 10th European software engineering conference and 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, 2005.

[15] S. Vasudevan. *High Level Static Analysis of System Descriptions for Taming Verification Complexity*. PhD thesis, The University of Texas at Austin, 2007.

[16] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In *Proc. of DATE.*, pages 545–548, 2010.

[17] V. M. Vedula, J. A. Abraham, J. Bhadra, and R. Tupuri. A hierarchical test generation approach using program slicing techniques on hardware description languages. *Journal of Electronic Testing: Theory and Applications*.

[18] S. Verma, K. Ramineni, and I. G. Harris. An efficient control-oriented coverage metric. In *Proc. of ASP-DAC*, pages 317–322, 2005.