Improving Design Verifiability by Early RTL Coverability Analysis

Kai-Hui Chang
Avery Design Systems, Inc.
Andover, MA, USA
changkh@avery-design.com

Chia-Wei Chang
National Central University
Jhongli, Taiwan
955401007@cc.ncu.edu.tw

Jie-Hong Roland Jiang
National Taiwan University
Taipei, Taiwan
jhjiang@cc.ee.ntu.edu.tw

Chien-Nan Jimmy Liu
National Central University
Jhongli, Taiwan
jimmy@ee.ncu.edu.tw

***Abstract*—Achieving high coverage is an important goal in design verification. Fixing coverability problems found at the verification stage, however, can require tremendous effort. To address this problem, we propose a flow for analyzing code and variable-toggle coverability at the early-RTL block-level stage. In addition, we devise a novel technique to analyze the coverability problems so that engineers can resolve the issues more efficiently. By identifying coverability problems at early RTL design stages, design verifiability can be improved, thus reducing the effort required at the verification phase.**

***Keywords*-Verification, Coverage, Satisfiability**

I. Introduction

Due to the dramatic increase in today's circuit complexity, design verification has become the bottleneck of a chip's development process. Traditionally, designers write their code with performance or area in mind and do not consider the verifiability of their code. Such an approach may create designs that contain numerous hard-to-verify points, prolonging the verification process and increasing the number of bugs that may escape verification. Therefore, it is desirable to find and fix verifiability problems early in the Register Transfer Level (RTL) development stage, especially at the block level before system integration. Typically, verifiability involves both controllability and observability aspects. In this work we only focus on the controllability issues.

To address the verifiability problem, several design-for-verification methods have been proposed. Mathur *et al.* proposed to use system-level models to assist the verification of the RTL code [9]. However, their methods cannot be applied when system-level models are unavailable. Liu *et al.* proposed to analyze the RTL itself and then insert *DFV points* for improving the controllability of hard-to-cover code [8], where a DFV point is a piece of logic that provides additional controllability for a specific portion of the design. Although their techniques can effectively improve design controllability, identifying hard-to-cover code in a design still remains challenging. One major reason is that their method for this purpose is based on sequential depth and certain heuristics, which may not correlate well with the real coverability problems encountered during verification. To better identify coverability problems, Ho *et al.* [5] created artificial coverage points for conditions in the RTL code and asked a formal tool to cover the coverage points with small time-out. If the tool returns "inconclusive" for a coverage point, then the condition associated with this point is considered to be hard-to-cover. The designer then changes the RTL code so that the condition can be covered more easily. One major limitation of this approach is that it only works if the tool used for determining coverability is the same tool that will be used for verification. To make sure the reported coverability problems are useful, it is desirable that the reported problems correlate well with what engineers use during verification. Given that constrained-random simulation is the prevalent verification method, it will be useful if hard-to-cover targets can be identified based on the number of random patterns required to cover the target instead of the runtime of one particular tool.

To address the above problem, we propose a flow to measure the coverability under two common coverage metrics: code reachability and variable toggleability. The flow is based on our new algorithms that can extract the Boolean condition when a conditional code block is entered or when a variable toggles. We then use a novel metric, *CovMet*, to measure the coverability of each coverage target with the extracted Boolean conditions. Once hard-to-cover targets are found, our innovative coverability analysis technique can then be applied to obtain the distribution of input patterns to cover the targets — such distribution can provide insightful information on how to address the coverability problems. Our experimental results show that the CovMet metric correlates well with random simulation, and our coverability analysis can provide insightful information for understanding how the coverability problem can be addressed.

The rest of the paper is organized as follows. Section II provides the background for understanding this paper. Our coverability measurement flow is described in Section III, and our coverability analysis technique is explained in Section IV. The experimental results are presented in Section V, and Section VI concludes this paper.

II. Background

In this section we first introduce several common verification practices, and then illustrate the And-Inverter Graph (AIG) structure that will be used in our coverability estimation methods. Finally, we describe a technique for

deriving evenly-distributed SAT solutions that will be used in coverability analysis.

A. Common Verification Practices

There are two commonly-used verification methods: simulation-based verification and formal verification.

Simulation-based verification evaluates a circuit using logic simulation. It is the mainstream method of verification because it is scalable and is easy to use. The most straightforwardly used method, called *direct test*, is to manually develop the input patterns that will be applied to the design. To automate test generation, constrained-random simulation was commonly adopted. This method constrains the input patterns based on the restrictions from the environment and can efficiently produce a large number of legal patterns. Since the patterns are not biased by engineers' intentions, unanticipated cases can be covered and bugs can be found. However, it can be difficult for pure-random patterns to cover corner cases. To address this problem, biased random simulation techniques [14], [15] have been developed. By changing the distribution of input patterns, designs can be verified more thoroughly. Furthermore, such methods can be applied to aid formal verification [12] after the circuit is fully analyzed.

Formal verification is based on rigorous mathematical reasoning, and thus it can prove or disprove the correctness of a circuit. Despite its strong proving power, formal methods are still not used extensively due to their scalability limitations. In recent years, the scalability and capability of formal methods have been dramatically improved, such as the work in [18].

One particular formal technique that will be used in this work is symbolic simulation. Compared with logic simulation, instead of simulating scalar and Boolean values, symbolic simulation allows symbols to be used as inputs. At each cycle, a Boolean expression, also called a *symbolic trace*, will be generated for each node in the design to represent the value of the node based on the symbols. Since a symbol can represent both 0 and 1 simultaneously, the trace exhaustively represents all the feasible inputs. Traditional symbolic simulators [1] can only handle gate-level circuits. To resolve this issue, Kolbl *et al.* [6], [7] enhanced symbolic simulation to handle RTL constructs such as delay statements. In this work, symbolic simulation is used to generate symbolic traces and will be used in Section III-B.

B. And-Inverter Graphs

An *And-Inverter Graph (AIG)* is a logic representation that became popular recently. Unlike some other data representations, such as *Binary Decision Diagrams (BDDs)*, the size of AIGs grows linearly with the size of the Boolean function, making AIGs scalable for handling large designs. However, the AIG for a function is not canonical, making certain Boolean manipulations less efficient than other logic representations. To solve this problem, Mishchenko *et al.* proposed *Functionally Reduced AIG (FRAIG)* [10] which is "semi-canonical". FRAIG ensures that each node in an AIG represents a unique function. They also proposed an efficient algorithm for converting AIGs to FRAIGs. AIGs can be used to represent both combinational and sequential circuits, and they are used in many applications such as formal verification and logic synthesis [18]. In this work, FRAIG will be used to derive our coverability metric, as we will show in Section III-C.

C. Deriving Evenly-Distributed SAT Solutions

Based on the Valiant-Vazirani theorem [13], Plaza *et al.* [11] proposed to use XOR-constraints to generate SAT solutions that are evenly distributed. In their technique, assume that a SAT instance f with variables $x_1, x_2, \ldots, x_n$ has solutions $v \in \{0,1\}^n$. They randomly pick an assignment $w \in \{0,1\}^n$ for constraining f with $(\sum_i v_i \cdot w_i) = 0$ in base-2 arithmetic. It effectively results in the constraint:

$$f \wedge (x_{i_1} \oplus x_{i_2} \oplus \cdots \oplus x_{i_j} \oplus 1) \tag{1}$$

where i_j represents the indices of x_{i_j} where w is 1. In [11], it is shown that if S_f is the set of all solutions of f, then the addition of constraints from k such random w vectors probabilistically reduces the solution space to $2^{-k}|S_f|$.

III. Measuring Design Coverability

Our methods for measuring design coverability are described in detail in this section. In this work, we focus on measuring the difficulty to enter each conditional block or to toggle the value of a variable. We call the logic we are analyzing, no matter it is code coverage or variable toggling, a *cover target*. We first present the overall flow for improving design verifiability. We then describe our algorithms for deriving the new metric to measure design coverability.

A. Our Design-for-Verification Flow

Our flow for improving design verifiability is shown in Fig. 1. The flow should be applied at early-RTL stages at the block level to find problems early on. In this flow, we first perform coverability measurement to identify hard-to-cover logic. If hard-to-cover targets were found, we perform hard-to-cover problem analysis to produce an input pattern distribution report so that the hard-to-cover problems can be characterized. If no pattern can be found to cover a target, we flag it as "non-coverable". The design or testbench can then be repaired according to the analysis report so that the hard-to-cover logic can be covered. To this end, if we found that covering the target requires specific inputs, direct tests or formal tools may be necessary. If the input patterns to cover the target cluster into a small region, then biasing the constrained-random testbench should be sufficient. For either case, adding additional control points into the design

always helps [8]. Once the design or testbench is repaired, coverability measurement is performed again to find more problems. This process repeats until no further coverability problems can be found.

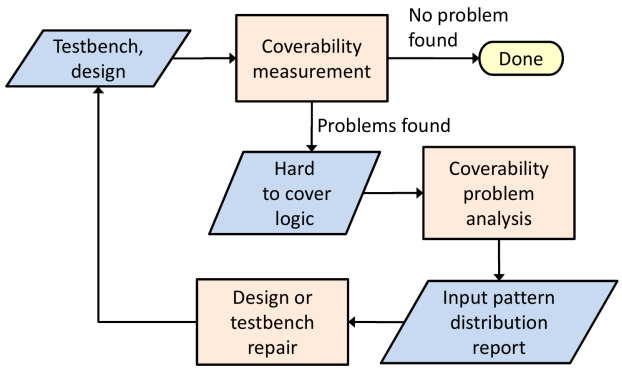

Figure 1. Our Design-for-Verification Flow.

B. Extract Boolean Conditions for Measuring Coverability

Since constrained-random simulation is the most commonly-used verification technique, when designing our coverability metric, called *CovMet*, we strive to ensure that it has high correlation with the number of random patterns explored in order to cover the target. To achieve this goal, we use symbolic simulation to generate a symbolic trace, in the form of a single-output Boolean function, that represents the condition for covering each target. Currently we inject a new symbol into each primary input at each cycle, and each register is initialized with a symbol. Under this setup, our coverability analysis will be based on unconstrained inputs and initial states. This analysis provides useful information regarding the design itself by pointing out the hard-to-cover logic caused by its logic function. If a testbench is available for generating legal inputs and the symbolic simulator can simulate the testbench, then our analysis will become testbench-aware and the coverability analysis results will be based on legal inputs, which will be even more accurate. Note that any formal method that can generate Boolean conditions for the cover targets can also be used. We chose symbolic simulation because it can generate Boolean functions for code-entering conditions more easily than other techniques. In addition, this flow can be applied to measure coverability of functional coverage points as long as the Boolean conditions for covering those points can be generated.

The algorithm for extracting code-covering conditions is shown in Fig. 2, which is based on the event-driven symbolic simulator in [3]. The inputs to the procedure are the event queue and the code statements to be executed. The events in the event queue are typically generated by the testbench. In line 1 of the algorithm an event is popped out from the event queue, and its symbolic condition is saved to *curr_sym_cond* as the current symbolic condition. In line

```
Proc extract_code_cover_condition(event_queue, statements);
1    event ← event_queue.pop();
2    curr_sym_cond ← event.sym_cond;
3    while execute statement triggered by event
4      if statement is a conditional block with condition cond
5        old_sym_cond ← curr_sym_cond;
6        curr_sym_cond ← curr_sym_cond AND cond;
7        statement.sym_cond ←
         statement.sym_cond OR curr_sym_cond;
8        if curr_sym_cond is not FALSE
9          execute statement;
10         curr_sym_cond ← old_sym_cond;
11     else if a new event nevent needs to be generated
12       nevent.sym_cond ← curr_sym_cond;
13       event_queue.add(nevent);
14     statement ← statement.next;
```

Figure 2. Algorithm for obtaining conditions for covering code targets (modified from [3]). Note that executing *statement* in line 9 can trigger multi-level condition evaluations and can be handled by executing lines 4-10 recursively.

3 we execute all the statements triggered by the event. If the statement is a conditional statement, we save the current symbolic condition to *old_sym_cond* in line 5. We then AND the current condition with the condition of the statement to create the new symbolic condition for executing the statement in line 6. To collect all the symbolic conditions that can cover *statement*, we create a field, *sym_cond*, for each *statement*. Whenever *statement* is covered, we OR the current symbolic condition with *statement.sym_cond* as shown in line 7. In this way, *statement.sym_cond* will be all the possible conditions for covering *statement*. The coverability of *statement* can then be measured based on its *sym_cond* using the metric that will be described in the next section. In line 8 we reject the execution of any statement whose symbolic condition is FALSE. In other words, we do not execute statements that cannot be entered. Lines 11 to 13 are used to handle the statements where new events are generated. For those statements, we save the current symbolic condition to the *sym_cond* field of the event and then add the event to the event queue.

The algorithm for covering toggle targets is shown in Fig. 3. It is executed at the end of the clock cycle to extract the condition for covering each variable *var* in the design *design*. In the algorithm, *var.sym_trace* is the symbolic trace produced by symbolic simulation that represents the value of the variable *var*. In lines 2-7 of the algorithm, we evaluate the symbolic trace *var.sym_trace* with random inputs using logic simulation. In other words, given the symbolic trace *var.sym_trace*, we assign random values to its inputs, and then use logic simulation to calculate its output. The value produced at the output is one possible value for variable *var*. We repeat this process N times and check if all the output values are the same, where N is arbitrarily chosen to

```
1   foreach var in design
2     for i ← 1 to 10
3       val ← simulate var.sym_trace with random inputs;
4       if (i = 1)
5         old_val ← val;
6       else if (old_val ≠ val)
7         break;
8     if (i = 10)
9       var.sym_cond ← var.sym_trace XOR val;
10    else
11      var.sym_cond ← null;
```

Figure 3. Algorithm for obtaining conditions for covering variable-toggle targets using logic and symbolic simulation.

be 10 in this work. If they are not the same, then *var* can easily have different values, so there is no need to check the toggle coverability of this variable. In this case, we set *var.sym_cond* to null in line 11 of the algorithm so that the coverability of the variable will not be checked. If all 10 output values are the same, then the toggle condition for *var* is *var.sym_trace* XOR *val*. We then save it to *var.sym_cond* in line 9.

C. CovMet: Our Coverability Metric

In the previous section, we extracted the condition for covering each target. Each condition is a symbolic trace that is essentially a single-output Boolean function. When the output of the function is 1, the target can be covered. Our goal is to calculate a number for our CovMet metric to quickly and effectively predict the difficulty for unbiased random simulation to cover the target. A naive way to calculate the coverability metric is to simulate a large number of patterns and count the number of 1s in the condition's *signature*, where a signature is a bit-vector of accumulated simulation values from the given inputs. This metric directly measures how difficult it is for random simulation to cover the condition. However, unless the number of patterns is huge, corner cases can be easily missed. For example, it is difficult to distinguish between a target that requires ten million patterns to cover and one that requires ten billion patterns unless billions of patterns have been simulated. The coverability problem is also addressed in the testing domain such as [4]. However, such techniques are difficult to apply at the RTL due to the complex logic relations among variables. For example, the accuracy of such techniques can reduce considerably when XOR operations are encountered.

In this work, we define our CovMet metric as the time spent on building the FRAIG for each target-covering condition. Our algorithm is based on the *fraig* implementation in [18], and the FRAIG structure is built bottom-up. Whenever a new node is added, random simulation is performed based on the patterns in its fan-in nodes, and the simulated values (also called its *signature*) are compared with other nodes. If the signature is unique, the node is functionally unique and can be added directly. Otherwise, a SAT solver is used to distinguish the node with another node that has the same signature. If a counterexample is found, then the node is functionally unique and is added to the FRAIG. The counterexample is then used as an additional pattern for random simulation to distinguish future nodes. Otherwise, these two nodes are functionally equivalent and are merged. In our implementation, the new FRAIG for a condition is always built upon the FRAIGs that have already been built for other targets. Given that symbolic conditions are often modified incrementally during symbolic simulation, opportunities to reuse existing FRAIGs are abundant. In addition, we record the runtime when building the FRAIG for each target and reuse the runtime whenever the node in the FRAIG is reused. In this way, FRAIGs for new conditions can be built incrementally instead of from scratch, thus considerably reducing the runtime for calculating our CovMet metric. An example is shown in Fig. 4.

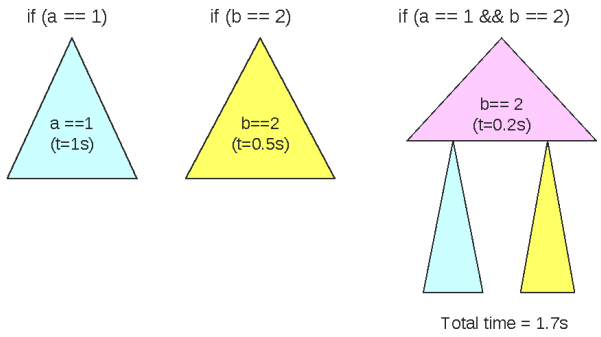

Figure 4. An example to show runtime and structure sharing when building FRAIG: the information for condition $a == 1$ and $b == 2$ is reused for $a == 1 \&\& b == 2$.

The reason why we designed the CovMet metric based on the runtime of building FRAIG is as follows.

1) Random simulation is used during the construction of FRAIG, which correlates well with our target — measuring the difficulty for random simulation to hit the target.
2) SAT is used to prove the equivalency of FRAIG nodes with the same signatures. This step solves the problem that the coverability of different targets cannot be measured if neither can be covered using the random vectors. In addition, the SAT solving step provides the power to distinguish between a target that is moderately difficult to cover and one that is extremely difficult to cover — SAT solvers tend to spend more time when the solution space is much smaller. It is possible that SAT solvers can "compress" the runtime such that a target requiring 10^6 more patterns to cover only requires 10X time to solve. However, this is necessary because without such compression the CovMet metric will take as long as simulating a huge

number of patterns.

3) The use of FRAIG logic structures and runtime information to be shared among different targets. Runtime can thus be reduced significantly. Otherwise, calculating the CovMet metric for two targets that are difficult to cover will always require solving two difficult SAT problems from scratch even if large portions of their logic overlap, which will be highly inefficient.

Due to the random nature of the algorithm, it is not guaranteed that the CovMet numbers will be exactly the same with different random seeds. However, statistically the CovMet number for the same target should be similar. As we will show in our experimental results, the CovMet metric correlates well with the number of random patterns required to cover a target.

The number of cycles simulated and analyzed affects the accuracy of our coverability analysis. Suppose that the initial state is unconstrained, if one cycle is simulated, then the coverability of each target will be based on the combinational logic between the target as well as the primary inputs and registers in its fan-in cone. If two cycles are simulated, then the constraints representing the sequential behavior of the design will be generated and analyzed as well, producing more accurate results. Since simulating each additional cycle tends to require extra runtime that may increase non-linearly, there is a trade-off between runtime and the quality of our analysis. Note that even though formal methods are used in calculating CovMet, scalability is not a major concern because our work targets early-RTL when blocks are still small. By fixing verifiability issues when the blocks are still being developed, verification difficulty at the system level can be considerably reduced.

IV. Coverability Problem Analysis

Once targets with coverability problems are identified, the next step is to find out why the targets are difficult to cover. The solutions to the coverability problem typically depend on the distribution of the inputs that can cover the target. In other words, it is useful to know the on-set distribution of the Boolean function that represents the covering condition of the target. In this section we propose a novel technique that can provide such analysis based on input pattern partitioning, distance calculation, and the use of statistics or data mining to interpret the result.

A. Input Pattern Partitioning

To analyze the distribution of input patterns that can cover the target, we need to generate those patterns first. Obviously, the number of all possible input patterns can be huge, making sampling necessary. Given that the target is hard to cover by random simulation, we use a SAT solver to generate the test patterns. The challenge here is how to generate "high-quality" or "representative" pattern samples. If we simply add the generated patterns as new constraints and then ask the solver to return another pattern, the returned pattern may be very similar to the original ones, making the sample patterns non-representative.

To address this problem, we utilize the Valiant-Vazirani theorem [11], [13] by adding XOR constraints to the SAT instance. The theorem states that each XOR constraint has a high probability of reducing the solution space by half, thus it can generate input patterns that distribute more evenly in the solution space. In this work we generate 32 patterns for each target for our analysis.

One may argue that if solutions can be found, then the target is already covered, and there is no need to perform the analysis. To this end, note that we perform coverability analysis at the block level while verification is carried out at the system level. Being able to cover the target at the block level does not mean it can be covered at the system level. By finding and fixing hard-to-cover logic at the block level, system-level verification will become easier.

B. Pattern Distribution Analysis

To analyze the distribution of the input patterns, we use a vertex to represent an input pattern, and add an edge between each pair of vertices: the weight of the edge is the *distance* between the two vertices. Distance can be calculated in several different ways. In this work we use the Hamming distance between the patterns based on their bit-level representations. In other words, the distance between two patterns is N if N bits are different. For example, if three input patterns, 010, 000, and 011 were returned, a graph as shown in Fig. 5 will be built.

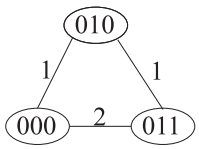

Figure 5. An example graph for pattern distribution analysis. The patterns are 010, 000 and 011. The Hamming distances between {010, 000}, {010, 011} and {000, 011} are 1, 1 and 2, respectively. The distances are used as the weights of the edges.

To improve the accuracy of the analysis, better distance metrics can be used. For instance, if the design under analysis is a processor, we may want to calculate the distance between two vertices based on the number of fields in the instructions that are different. In this way, the distance metric can better capture the relation between different input patterns. For example, two instructions with different OP codes will always have distance 1 instead of the number of bits that are different in their instruction-set encoding. Exploring different distance metric for different types of circuits is our future work.

After the graph is built, we can then derive the distribution of the input patterns by analyzing the distances between the vertices. One way to achieve this goal is to use data-mining to detect clustering of vertices. In this work we calculate

the average, minimal, maximum and standard deviation of the distances. As we will show in our experimental results, these numbers are useful for determining the distribution of the input patterns. Applying data-mining methods to analyze the graph is also our future work.

C. Implementation Insights

When conducting our experiments we observed that runtime may increase dramatically with each additional XOR constraint after a certain number of XOR constraints have already been added. This is expected because the SAT solver has to search for solutions under more constraints. Since more XOR constraints can better partition the solution space, there is a trade-off between runtime and the accuracy of our analysis. To provide enough accuracy for our analysis without incurring large runtime penalty, we use the following heuristic. (1) Keep adding XOR constraints and measure the runtime for the SAT solver to return the next solution. (2) When the runtime increases by N times between two successive SAT calls, we abort the search. N is determined empirically in this work.

V. Experimental Results

The Bug UnderGround project [17] from Michigan provides two processor designs, Alpha and DLX, that contain various bugs. Both Alpha and DLX are 32bit 5-stage pipelined processors. Alpha is supports a subset of the Alpha instruction set and DLX supports a subset of MIPS instruction set. In this work we chose the DLX processor from the project because its different bugs require different levels of efforts to cover. By comparing our results with the number of random patterns used to cover the bugs, we can measure how well our metric correlates to real coverage difficulty. The machine we used was a Dell PowerEdge 2900 (2GHz Xeon processor with 48 Gbytes of memory) running CentOS Linux 5.5. We implemented our algorithms on a commercial symbolic simulator [16] whose implementation detail can be found in [2], and the ABC package from University of California at Berkeley [18].

A. CovMet Accuracy Evaluation

In this experiment we calculated the CovMet value for each DLX bug-triggering condition and compared the results with the number of random patterns used to trigger each bug. For comparison, we also show the number of AIG and FRAIG nodes of the Boolean function built for each bug-triggering condition. We symbolically simulated the design for 7 cycles because it takes up to 7 cycles for an instruction to retire from the pipeline. For designs whose sequential depth is unknown, the user should choose the number of cycles to analyze based on runtime and accuracy trade-off: analyzing more cycles provide more accurate analysis at the expense of longer runtime. The bug-triggering conditions are then derived using the algorithm shown in Fig. 2. The results are summarized in Table I, in which the correlation coefficients are also shown. The runtime of all benchmarks was within 1 minute. Note that the number of patterns to trigger each bug can be considerably different from [2], [14] because we were measuring the number of patterns to trigger the bug, while [2], [14] measured the number to observe the bug at primary outputs. We also observed that several bugs could not be covered by random simulation using one million patterns, and it is difficult to distinguish the coverability of those bugs unless more patterns are simulated to a point where at least one pattern covers all the bug-triggering condition. On the other hand, our coverability metric for all the bugs could be successfully calculated in a minute, suggesting the efficiency of CovMet calculation.

Table I
THIS TABLE SHOWS THE NUMBER OF RANDOM PATTERNS USED TO TRIGGER EACH BUG, THE NUMBER OF AIG/FRAIG NODES BUILT FOR THE BOOLEAN FUNCTION TO TRIGGER THE BUG, AND OUR COVMET NUMBERS. "CORR." IS THE CORRELATION COEFFICIENT BETWEEN EACH METRIC AND THE NUMBER OF RANDOM PATTERNS.

Bug ID	#AIG node	#FRAIG node	CovMet	#Random patterns
Bug1	49932	42365	0.002316	20
Bug2	51117	46403	0.001435	35
Bug3	39499	21522	4.473474	62
Bug4	117796	33167	1.532637	2
Bug5	60311	43660	6.575513	61
Bug6	239719	33722	6.753490	59239
Bug7	24852	9506	0.872565	87
Bug8	159297	34516	1.709590	6824
Bug9	853853	48635	9.119023	7
Bug10	848088	45183	8.254419	220
Bug11	5430	2653	0.592229	5
Bug12	11754	5937	0.671420	1
Bug13	219844	34686	1.765894	79
Bug14	275114	34737	1.757642	47
Bug16	278783	34846	1.748167	47
Bug17	224859	34141	6.948297	77
Bug18	226892	35372	7.021435	66
Bug19	39161	25570	4.666351	5950
Bug21	64323	43210	6.912815	95
Bug23	60049	42675	6.581663	9785
Bug24	138514	25535	11.354756	>1M
Bug25	177553	35486	31.203642	>1M
Bug26	125687	34125	6.936416	>1M
Bug27	42720	10468	20.623046	>1M
Bug28	949335	47403	2.294788	235878
Bug30	75820	20934	4.621573	1215
Bug32	42773	7823	1.385160	13660
Bug35	1119251	52362	9.281496	9941
Bug36	238584	31332	1.868086	1
Bug37	42763	10462	1.436329	>1M
Bug38	318769	35792	6.867821	>1M
Bug39	21332	14534	1.029208	>1M
Bug40	224297	33937	7.276143	79
Corr.	-0.14	-0.25	0.48	1.0

From the result, we can see that measuring coverability

using the number of AIG or FRAIG nodes is not as accurate as CovMet because their correlation coefficients with the number of random patterns are closer to 0. That the correlation coefficient between our metric and the number of random patterns is 0.48 suggests that our metric predicts whether or not the target will be hard-to-cover in random simulation pretty well. In the future we plan to improve this number by considering the the aforementioned compression effects.

B. Pattern Distribution Analysis

In this section we evaluate our pattern distribution analysis techniques to see how well they correlate to the real cause of the coverability problems.

To provide some insights into our analysis technique, in the first experiment we performed our analysis on a 12-bit multiplier. Four targets were designed in this experiment. The first one solves for two values that multiplies to 65531. Since there are only a few possible solutions, it represents the case where formal methods or direct tests are necessary. The second one can be covered when the multiplied number is larger than 8388608, and the third one can be covered when the multiplied number is smaller than 5000. These two cases represent the situation where biased-random simulation can be useful. The fourth one can be covered when the multiplied number is a multiple of 5. It represents the case where unbiased-random simulation is sufficient. In this experiment, we ran input space partitioning 32 times to get 32 samples for each target. For each sample, 25 random XOR constraints were added. We then computed the Hamming distances between all 32 pairs of samples and then calculated the minimal, maximal, average and standard deviation of all the distances. The largest runtime for all the samples was 19 seconds, and the results are summarized in Table II.

Table II
STATISTICS OF HAMMING DISTANCES AMONG 32 SAMPLE POINTS FOR EACH TARGET.

Target ($result = a \times b$)	Avg.	Min.	Max.	Std. dev.
1 (result=65531)	8.82	0	15	5.77
2 (result>8388608)	11.76	4	19	2.51
3 (result<5000)	9.69	1	20	3.27
4 (result mod 5 == 0)	11.98	6	19	2.48

From the results, we can see that when only a few solutions exist, such as target 1, the average distance is small and the standard deviation is large. This is because most solutions cluster within very small regions, thus reducing the average distance between each solution. However, the distance variance to another region is high, resulting in large standard deviation. Note that we do not exclude repetitive solutions because this situation indicates that the number of possible solutions is probably small. When the solutions start to spread out, such as target 3, the average distances between solutions start to increase, but the standard deviation begins to decrease. In the case that solutions distribute almost evenly in the input space, such as target 4, the average distance will be large and the standard deviation will be small. This is because the distance to another solution increases due to the lack of clustering, while the distance variance decreases due to the even distribution of solutions.

In summary, when the average is low and the standard deviation is high, formal methods are probably necessary to find a solution. When the average is high and the standard deviation is low, unbiased random simulation is probably enough to find a solution. For those in between, biased random simulation is probably necessary.

In the second experiment, we selected the bugs that needed more than 1000 patterns to cover and then performed our distribution analysis on those bugs. The distance calculation was based on Hamming distance, and the analysis was the same as the multiplier. The total runtime of the analysis was 6.5 hours, and the results are summarized in Table III.

Table III
STATISTICS OF HAMMING DISTANCES AMONG 32 SAMPLE POINTS FOR EACH BUG.

Bug ID	Avg.	Min.	Max.	Std. dev.
Bug6	76.1	32	123	15.33
Bug8	67.31	20	129	17.4
Bug19	68.14	35	98	11.75
Bug23	85.37	39	148	17.79
Bug24	57.68	15	116	16.75
Bug25	78.33	28	117	15.02
Bug26	73.76	23	137	17.3
Bug27	67.62	39	97	8.79
Bug28	73.81	23	120	16.3
Bug30	69.34	27	95	11.51
Bug32	61.02	25	96	12.62
Bug35	71.28	26	120	15.85
Bug37	57.4	22	86	11.43
Bug38	74.63	31	149	20.84
Bug39	58.03	24	98	15.72

To show the effectiveness of our analysis, we analyzed the cases with the largest and the smallest average distances. Bug23 is the one with the largest average distance. The bug triggering condition is "*rs*=9" or "*rt*=9 and *op*=LW (load word)", where *rs* and *rt* are register fields in the instruction. Obviously, there are many different ways to trigger the bug, so the solution distribution is even, which results in large average distance. Bug37 is the one with the smallest average, and its bug triggering condition is "*rd* = *rt* one cycle earlier, *rd* = *rs* 2 cycles earlier, and *op* = SPECIAL at the current cycle and SW at the previous cycle". Obviously, triggering this bug requires very specific conditions, thus producing clustering of solutions that make the average distance small.

C. Case Studies

We applied our coverability analysis flow to the DLX design. We started from random states and analyzed design coverability for 7 cycles for both code and toggle targets. For code targets, 245 conditional code blocks were analyzed, and runtime was 16h46m. The condition that took the most time used 818 seconds for building the FRAIG structure. This condition controls the signal that indicates whether the instruction register at stage 2 should be stalled, and the code it controls can only be entered under specific sequences of branching instructions. Given that pipeline stall logic is one of the most complicated circuitries in this design, our analysis pointed out the hard-to-enter code that the designer should be aware of.

For toggle coverage, we analyzed 58 word-level registers, and runtime was 4h21m. The register that required the most time for building its FRAIG took 1930 seconds and is the stall signal from the instruction decoder bypass block. Similar to the result in code verifiability analysis, making this register having value 1 also requires specific sequences of branching instructions, which is consistent with the observation that this is one of the most complicated circuitries in this design. This result further demonstrates the effectiveness of our methodology in finding hard-to-cover code in the design. For these two case studies, we used a formal tool to generate test patterns to cover the targets.

VI. Conclusion

In this work we proposed algorithms and a new metric to measure the coverability of RTL code at early design stages. In addition, we devised a new technique for better understanding the coverability problem by analyzing the input distribution for covering the target. Our empirical evaluations show that the coverability metric correlate to the results from random simulation well, and our coverability analysis can provide insightful information for designers to resolve the problems. By analyzing and fixing coverability problems early in the design stage, design coverage can be improved at the verification phase, resulting in shorter verification time and better design quality.

Acknowledgment

This work was supported in part by National Science Council under grants 99-2221-E-002-214-MY3, 99-2923-E-002-005-MY3, and 100-2923-E-002-008.

References

[1] R. E. Bryant, "Symbolic Simulation – Techniques and Applications," *DAC'90*, pp. 517-521.

[2] H.-Z. Chou, I.-H. Lin, C.-S. Yang, K.-H. Chang and S.-Y. Kuo, "Enhancing Bug Hunting Using High-Level Symbolic Simulation", *GLSVLSI'09*, pp. 417-420.

[3] H.-Z. Chou, K.-H. Chang and S.-Y. Kuo, "Optimizing Blocks in an SoC Using Symbolic Code-Statement Reachability Analysis", *ASPDAC'10*, pp. 787-792.

[4] L. H. Goldstein and E. L. Thigpen, "SCOAP: Sandia Controllability/Observability Analysis Program", *DAC'80*, pp. 190-196.

[5] R. C. Ho, M. Theobald, M. M. Deneroff, R. O. Dror, J. Gagliardo, D. E. Shaw, "Early Formal Verification of Conditional Coverage Points to Identify Intrinsically Hard-to-verify Logic", *DAC'08*, pp. 268-271.

[6] A. Kolbl, J. Kukula and R. Damiano, "Symbolic RTL simulation," *DAC'01*, pp. 47-52.

[7] A. Kolbl, J. Kukula, K. Antreich, and R. Damiano, "Handling Special Constructs in Symbolic Simulation," *DAC'02*, pp. 105-110.

[8] C.-N. J. Liu, I.-L. Chen and J.-Y. Jou, "An Efficient Design-for-Verification Technique for HDLs", *ASPDAC'01*, pp. 103-108.

[9] A. Mathur and V. Krishnaswamy, "Design for Verification in System-level Models and RTL", *DAC'07*, pp. 193-198.

[10] A. Mishchenko, S. Chatterjee, R. Jiang, R. Brayton, "FRAIGs: A Unifying Representation for Logic Synthesis and Verification", ERL Technical Report, EECS Dept., U. C. Berkeley, March 2005.

[11] S. M. Plaza, I. L. Markov, and V. Bertacco, "Random Stimulus Generation using Entropy and XOR constraints", *DATE'08*, pp. 664-669.

[12] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, K. Keutzer "A Functional Validation Technique: Biased-Random Simulation Guided by Observability-Based Coverage", *ICCD'01*, pp. 82-88.

[13] L. Valiant and V. Vazirani, "NP is as easy as detecting unique solutions", *Theor. Comput. Sci.*, pp. 85-93, 1986.

[14] I. Wagner, V. Bertacco, T. Austin, "StressTest: An Automatic Approach to Test Generation Via Activity Monitors," *DAC'05*, pp. 783-788.

[15] H.-J. Wunderlich, "Multiple distributions for biased random test patterns", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Jun. 1990, pp. 584-593

[16] Avery Design Systems Inc., http://www.avery-design.com

[17] Bug UnderGround, http://bug.eecs.umich.edu

[18] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/∼alanmi/abc/abc.htm