# Facilitating Unreachable Code Diagnosis and Debugging

Hong-Zu Chou[†], Kai-Hui Chang[‡], and Sy-Yen Kuo[†]
[†]Electrical Engineering Department, National Taiwan University, Taipei, Taiwan
[‡]Avery Design Systems, Inc., Andover, MA, USA
sykuo@cc.ee.ntu.edu.tw

**Abstract— Code coverage is a popular method to find design bugs and verification loopholes. However, once a piece of code is determined to be unreachable, diagnosing the cause of the problem can be challenging: since the code is unreachable, no counterexample can be returned for debugging. Therefore, engineers need to analyze the legality of nonexistent execution paths, which can be difficult. To address such a problem, we analyzed the cause of unreachability in several industrial designs and proposed a diagnosis technique that can explain the cause of unreachability. In addition, our method provides suggestions on how to solve the unreachability problem, which can further facilitate debugging. Our experimental results show that this technique can greatly reduce an engineer's effort in analyzing unreachable code.**

## I. Introduction

Code coverage is a verification technique for identifying reachable code statements and is used frequently to find bugs in designs and testbenches. However, traditionally code coverage analysis is often performed with logic simulation. Therefore, corner cases may be missed and no proof on reachability can be provided. To address this problem, symbolic code reachability analysis can be used [7]. Due to the formal nature of this technique, the reachability of code statements can be exhaustively examined and accurately determined.

Although the process of finding unreachable code has been highly automated, diagnosing the root cause of unreachability is still a tedious and time-consuming process. One major reason is that unlike most bugs for which verification tools can provide counterexamples, the fact that a statement is unreachable means no counterexample can be provided. The lack of stimulus to expose the unreachability problem makes debugging especially difficult because designers will have to imagine new execution paths to make the code reachable. The lack of counterexamples also makes the deployment of existing automatic error diagnosis methods [6, 15] difficult because there is no trace to utilize and no expected values to solve for. Another reason is that debugging hardware code can be much more difficult than debugging software code because software code is executed sequentially[1] while the execution traces in hardware designs are parallel in nature. As a result, software dead code analysis techniques cannot be utilized [14]. In addition, the conditions that caused a piece of code to be unreachable can come from remote blocks that seem to be irrelevant and can be extremely difficult to identify.

To facilitate unreachability diagnosis, we propose a new technique based on symbolic simulation to identify key variables that caused the unreachability and provide execution paths that can reach the target code. Our main contribution is an innovative way to modify a symbolic simulation algorithm so that the algorithm has the freedom to explore execution paths that do not exist before. This innovation addresses one major problem that prevents traditional synthesis-based model-checkers from being able to perform unreachability diagnosis: the logic may not be in the model at all because the code is proven to be dead. Our second contribution is to intelligently leverage state-of-the-art formal error-diagnosis methods to analyze the symbolic condition when entering previously unreachable code. In this way, we can provide diagnosis to explains code unreachability. Our experience and case studies show that our techniques can considerably narrow down the unreachability problem and reduce designer's debugging effort. As Section III-C suggests, this diagnosis technique can also help solve difficult formal verification problems related to deep sequential circuits and highly configurable designs.

The rest of the paper is organized as follows. Section II provides necessary background to understand our work. In Section III we conduct an analytical study on the unreachability problem and propose several potential applications. Our unreachability diagnosis techniques are described in detail in Section IV. Experimental results are shown in Section V, and Section VI concludes this paper.

## II. Background

Our unreachability diagnosis method is based on symbolic simulation and hardware error diagnosis techniques. In this section, we first provide an overview of existing hardware error diagnosis approaches, and then briefly describe the characteristics of symbolic simulation.

### A. Hardware Error Diagnosis Techniques

Error diagnosis techniques for hardware designs have been extensively investigated over the past decade [3, 15, 18, 19, 20]. Early work for error diagnosis mostly focused on the gate level and was limited to single errors [18, 19]. Recently, error diagnosis is enhanced to handle RTL designs and multiple errors. For instance, Boppana et al. [3] exploited hierarchies available in RTL designs to locate design errors. Shi et al. [20] employed a software-analysis method to reduce error space. Jiang et al. [15] proposed an approach that reduces the error candidates by estimating the correctness of potentially erroneous statements.

Error diagnosis and debugging have also been addressed by formal SAT-based solutions [1, 2, 6, 12, 21]. To this end, Smith et al. [21] and Ali et al. [1] proposed SAT-based error diagnosis for sequential circuits: they use SAT to check whether the correct output can be generated by changing a limited number of gates in the erroneous circuit. Ali et al. [2] also proposed to use the hierarchical nature of designs to improve the performance and quality of fault diagnosis. Recently, Chang et al. [6] adopted this concept and proposed an efficient technique that can handle RTL statements. To address the problem that large potential sites are returned by probabilistic techniques, Fey et al. [12] proposed a property-based approach that can greatly improve the accuracy of error diagnosis.

[1]Debugging parallel programs is more difficult and is similar to debugging hardware code.

978-1-4244-7514-8/11/$26.00 ©2011 IEEE

### B. Symbolic Simulation

Symbolic simulation is a formal verification method that works particularly well with traditional simulation-based methodologies. Unlike logic simulation that only simulates scalar inputs, symbolic simulation uses Boolean variables, also called symbols, as inputs. It then produces Boolean expressions, also called symbolic traces, as outputs. Since each symbol represents both 0 and 1, symbolic simulation can evaluate all possible inputs simultaneously.

Traditional symbolic simulators [5] only support gate-level netlist representations, which are not flexible enough to handle today's complex designs and testbenches. To address this problem, Kolbl et al. [17, 16] introduced symbolic RTL simulation which is capable of handling RTL constructs and delay statements. Compared to gate-level symbolic simulation, RTL symbolic simulation is intrinsically a combination of software and hardware verification techniques, thereby enabling several unique advantages for design verification such as code-statement reachability analysis [7]. Take Figure 1(a) as an example, assume $op$ is unknown (denoted as 1'bx), logic simulation will generate inaccurate reachability report due to X-pessimism [4]. On the other hand, symbolic simulation replaces $sum$, $a$, $b$, $op$ with symbols $s_{sum}$, $s_a$, $s_b$, $s_{op}$, respectively. As shown in 1(b), $s_{sum}$ will then have a Boolean expression as its value instead of a scalar value. Therefore, it can produce more accurate reachability report compared with logic simulation, as shown in 1(c).

(a) if ($op$) $sum = a - b$; ..... $S1$
else $sum = a + b$; ..... $S2$

(b) Symbolic trace: $s_{sum} = (s_{op})\ ?\ (s_a - s_b) : (s_a + s_b)$ ;

(c) Code reachability report (assume $op$= 1'bx):
$S2$ can be reached by logic simulation.
$S1$, $S2$ can be reached by symbolic simulation.

Fig. 1. Code reachability analysis using logic simulation and symbolic simulation when X exists in the design. Due to X-pessimism, logic simulation can only reach $S2$ with respect to the Verilog standard, while symbolic simulation produces correct reachability results.

## III. Analysis of the Unreachability Problem

In this section, we first formulate the unreachability problem. We then studied several industrial designs and analyzed the cause of unreachability in these designs. Finally, we provide several potential applications that can benefit from code unreachability diagnosis.

### A. Problem Formulation

Unreachable code is often associated with errors and should be corrected. The main objective of our code-statement unreachability diagnosis is to find the cause of such errors and fix them. The diagnosis problem is formulated as follows. Given a design, a testbench and a list of unreachable code statements obtained from formal reachability analysis [7], we seek to find the cause of unreachability and provide a set of suggestions on how to fix the unreachability problem. Note that in our formulation, unreachability can be caused by design bugs, testbench bugs or improper testbench configurations.

### B. Common Causes of Unreachability

In order to design useful methods for unreachability diagnosis, we analyzed several industrial designs to identify the root causes of their unreachability. We found that the reasons can be broadly classified into four major types: hardware bugs, design modalities, testbench errors, and reachability analysis limitations. Hardware bugs are often due to incorrect use of predicates in conditional branches such as conflicting conditions. These bugs are usually serious because the circuit's behavior would not match the designer's intention when such bugs exist — few people put dead code into a design on purpose. Other common hardware bugs include obsolete code that has not been removed, such as unused tasks and functions.

From our analysis, we found that the most common types of testbench errors that result in code unreachability are incorrect operating modes and over-constrained rules. For example, the design is constrained to run in mode 1 but the designer is checking the code that is written for mode 2. Unreachability due to reachability analysis limitations is often caused by insufficient verification depth. For example, the number of simulated cycles may be too small for a counter to assume a value that allows certain code to be reached, such as code that detects the buffer full condition. Although such problems are not design bugs, they limit the power of formal tools and reduce the thoroughness of verification.

### C. Potential Applications

In this section, we describe three applications that can benefit from our unreachability diagnosis.

**Reducing debugging effort:** Existing verification tools cannot provide counterexamples for unreachable code[2], forcing designers to trace nonexistent execution paths in order to find where the bugs come from. Obviously, such manual work is time-consuming and inefficient. Our unreachability diagnosis method can identify key variables that designers should look at and automatically generate a set of suggestions to reach the target code. By modifying the testbench or design to produce the suggested values for those key variables, the unreachability problem can be solved, making debugging much easier.

**Facilitating formal verification:** Formal verification techniques based on Bounded Model Checking (BMC) can provide comprehensive results for the unrolled cycles. Due to its bounded nature, however, BMC has difficulty verifying properties with large sequential depth such as the FIFO full condition. One way to solve this problem is to abstract certain key variables, such as counters, to allow formal analysis of related properties [10]. Currently, such abstraction is often performed manually by designers. With our unreachability diagnosis method, formal tools can perform automatic abstraction on the variables that we identified to cause unreachability and prove properties that could not be proved before.

**Setting reasonable code coverage goals:** In order to support design reuse, a circuit may have multiple modes and configurations so that it can be used in different applications. When verifying a specific configuration, a substantial amount of unused code may be discovered because the code may be written for other modes. However, designers cannot simply ignore the unreachability report because the dead code may be caused by other problems. By giving the configuration registers more freedom in choosing their values, we can quickly identify unreachable code that is due to the current configuration and remove them from coverage targets. Designers can then focus on the code coverage for the rest of the design.

[2]Conventional formal tools, such as model checker, cannot be used for unreachability diagnosis because they are synthesis-based. Since the code is dead, most likely the logic associated with the code will not be in the model at all.

## IV. Unreachability Diagnosis

In this section we first describe how we modify symbolic simulation for unreachability diagnosis, and then show the procedure to perform the diagnosis using SAT solvers. Finally, we present how to generate counterexamples that can make the unreachable code reachable. Note that our unreachability diagnosis is considerably different from the error diagnosis techniques used in [1, 6] in that: (1) traditional error diagnosis requires the expected (correct) values at primary outputs or key variables to be known, while we don't; and (2) traditional techniques are mostly synthesis-based and do not take advantage of software execution features, while ours modify the symbolic simulation algorithm and is closer to software debugging. Most importantly, traditional techniques insert error modeling constructs *after* synthesis; therefore, these techniques don't have the ability to explore code execution paths that do not exist in the design. On the contrary, our use of multiplexers (MUXes) changes symbolic simulation itself; therefore, our method can explore previously-nonexistent execution paths.

### A. Symbolic Simulation for Unreachability Diagnosis

The unreachability diagnosis problem in our work is represented with (1) a design containing a list of unreachable code statements identified by reachability analysis techniques such as [7, 9]; and (2) a testbench or a set of input patterns under which the code statements are unreachable. In this work, we modify the event-based symbolic simulation algorithm described in [7, 16] in order to perform unreachability diagnosis. The reason why we use symbolic simulation algorithm instead of other path-oriented analysis techniques is that it can explore all execution paths for the target code statements under the given input constraints. When the target code statements are unreachable, there must be one or more conflicts in the symbolic condition to execute the target code.

Our modified symbolic simulation algorithm is shown in Figure 2, which is based on Figure 2 in [7]. In order to identify key variables that cause conflicts in symbolic execution, we select certain variables, called *liberated variables*, to give them more freedom in choosing their values when they are accessed in symbolic simulation. Our algorithm will then mark a subset of the liberated variables as key variables that cause the unreachability. Since the designer may not know where the problem is from, in practice we typically make all design variables, except the clock and reset signals, liberated variables. To reduce diagnosis time, hierarchical approaches, such as [2], can also be applied. As line 4 shows, the liberated variables are implemented by MUXes. More specifically, a conditional assignment is created to determine whether a variable $v_i$ contributes to the unreachability and should have its behavior changed or not. When the select line $V_{i\_s}$ is 0 (note that we use uppercase names for symbolic traces and lowercase for variables), the symbolic trace returned by symbolic simulation uses the original trace $V_i$, which means symbolic conditions will not be changed by $v_i$. In other words, variable $v_i$ does not contribute to the unreachability and has its behavior unchanged. On the other hand, when $V_{i\_s}$ is 1, symbolic trace is replaced with a free symbol $V_{i\_free}$ that can take any value. In this way, there is an opportunity to eliminate the conflicts in symbolic conditions so that execution paths that could not be taken before can now be taken, which can then allow symbolic simulation to explore new execution paths. When a previously-unreachable code statement is reached, we can then use the method that we will show in the next section to perform unreachability diagnosis.

**Procedure *symbolic_simulation_for_unreachability_diagnosis()***

```
01. event = event_queue.pop();
02. curr_sym_cond = event.sym_cond;
03. while execute statement triggered by event
04.     whenever a liberated variable, v_i, is accessed,
        replace returned symbolic trace V_i with V_i_s ? V_i_free : V_i;
05.     if statement is a conditional block with condition cond
06.         curr_sym_cond &= cond;
07.         do not execute statement if curr_sym_cond is proven to be 0;
08.         if code_to_be_diagnosed is reached
09.             unreachability_diagnosis(curr_sym_cond);
10.     else if leaving conditional block with condition cond
11.         restore curr_sym_cond by removing cond as constraint;
12.     else if a new event nevent needs to be generated
13.         nevent→sym_cond = curr_sym_cond;
14.         event_queue.add(nevent);
15.     statement = statement.next;
```

Fig. 2. Modified event-driven symbolic simulation algorithm for unreachability diagnosis. Lines with boldfaced numbers contain our changes.

### B. Diagnosis and Counterexample Generation

To narrow down the cause of unreachability, cardinality constraints, shown in Figure 3, are added to restrict the number of select lines that can be asserted simultaneously. More specifically, we use $n_e$ to denote the number of candidate variables that could contribute to the unreachability, and this number should be as small as possible so that the problem can be localized to a few key variables. Our algorithm for unreachability diagnosis is shown in Figure 4. In the algorithm, $n_e$ is initialized to 1 and we gradually increase the number until we find valid diagnoses. Since we enumerate all the reasons that can cause the unreachability, the designer can consider all possible scenarios to solve the problem. To facilitate debugging, we rank the results according to their distance to the primary inputs of the symbolic trace based on the observation that solutions closer to those inputs are typically more useful. Note that diagnosis time can increase rapidly when $n_e$ increases. However, this is not an issue in practice because most unreachability problems can be attributed to a few key variables.

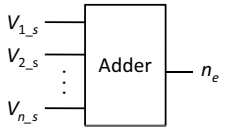


Fig. 3. Cardinality constraints for unreachability diagnosis.

**Procedure *unreachability_diagnosis (sym_cond)***

```
01. for (n_e = 1; sym_cond is UNSAT; n_e++)
02.     while (sat_solving(sym_cond) == SAT)
03.         report variables associated with the asserted select lines
            as cause of unreachability;
04.         report values in free variables associated with the asserted
            select lines as counterexample;
05          add negation of asserted select lines as new clauses;
```

Fig. 4. Unreachability diagnosis for a given symbolic condition.

In line 2 of the algorithm, when SAT found a solution, variables with their select lines asserted are those contribute to the unreachability: by changing their behavior using the values returned in the free symbols, the code will become reachable. Therefore in line 3 we report such variables as cause of the unreachability problem, and the values reported in line 4 are the counterexample. In order to find all possible diagnoses, in line 5 we add the negation of asserted select lines as new clauses so that the same solution will not be returned again.

### C. Implementation Insights

The selection of liberated variables affects both the performance of symbolic simulation and the accuracy of unreachability diagnosis. In general, variables (registers or wires) used in a design are good candidates of liberated variables. Since too many liberated variables may reduce our diagnosis performance due to unnecessarily large search space, one can also restrict liberated variables to dominating registers only. It is important to note that certain types of variables, such as clocks, resets and loop index variables, should not be selected as liberated variables because they are typically unrelated to code unreachability. In addition, giving them freedom to choose different values will create unnecessary overhead for symbolic simulation. For example, making the clock a liberated variable will cause symbolic simulation to trace all possible combinations of clocks, which will be highly inefficient.

RTL symbolic simulation performs operations at the word level. As a result, only one MUX is added to a multi-bit variable, which can considerably reduce the search space of SAT solvers because all bits share the same select line. On the other hand, each bit still has its own free symbol to ensure maximum flexibility in resolving the conflicts that cause the unreachability. Another way to reduce SAT search space is to reduce the number of inserted select lines. Since in hardware, a variable typically has only one value at a specific time, we can use the same select line to control the MUXes inserted at the same time step. However, a variable can have multiple values in a testbench, and one of them could be the cause of conflict conditions. Therefore, we have to use separated select lines to control the MUXes. Since our method is generic, it can be applied to all types of HDL constructs, including behavior testbench, as long as the constructs can be symbolically simulated.

## V. Experimental Results

We chose a DLX processor from the BugUnder Ground project [23] as the benchmark in this paper because it is one of the few publicly-available designs that contains non-trivial unreachable code due to design errors. DLX [13] is a 32-bit RISC microprocessor with 5-stage pipeline. The BugUnder Ground project provides a DLX implementation with 40 manually inserted bugs; however, due to conflicting conditions, it is found that 6 of the bugs actually can never be triggered [8]. In other words, the code statements for triggering the bugs can never be reached. Since the other 34 bugs can be triggered, we only use the 6 "dead bugs" to evaluate our unreachability diagnosis. A detailed description of the bugs are listed in Table I.

In addition to academic benchmarks, we also applied our diagnosis techniques to several blocks from industrial designs, including a block in a multimedia design and a high-speed I/O interface. We found that the diagnosis results can often greatly reduce debugging time. Since the designs are confidential, we only report the design sizes and the results of our unreachability analysis technique.

We implemented our algorithms using a commercial symbolic simulator called Insight [22] that supports behavior and RTL Verilog. In our use model, designers first perform formal reachability analysis to find unreachable code. Next, they choose the code that needs to be diagnosed by specifying its file name and line number. Liberated variables are then selected using a user interface similar to VCD dump. Finally, symbolic simulation is performed using the same environment as the one used for reachability analysis for unreachability diagnosis. In our experiments, runtime and memory usage were calculated for the whole process, including symbolic simulation and the SAT solving part of unreachability diagnosis. The machine we used was a Dell PowerEdge 2900 (2GHz Quad-Core Xeon, 48 Gbytes main memory) running Linux Fedora Core 8. MiniSat [11] complied in 64-bit mode was used as the SAT solver for unreachability diagnosis.

### A. Diagnosis Example

Evaluating automatic debugging methods is often difficult because the quality of diagnosis is highly subjective — the same diagnosis can help one engineer while being totally confusing to another one. As a result, the quality of diagnosis is often hard to quantify. Therefore, in this subsection we use an example to show what our diagnosis report looks like and explain how it can facilitate designers to debug the unreachability problem. The example is from DLX BUG22. This bug [23] supposedly should cause the destination register to be wrong under certain conditions, as shown in Figure 5(a). However, we found that the bug can never be triggered. To debug the problem, we performed unreachability diagnosis, and one of the diagnosis is shown in Figure 5(b). In this case, instructions are generated at "negedge clk" and are propagated to the next stage at "posedge clk". The report shows that the bug can be triggered by changing variable IR2 (instruction register at the 2nd-stage pipeline) with the suggested values. However, the report shows that in order to produce "RDaddr5==5'd7" and "IR5[`op]==`ADD" at time 550, the values in IR2 has to be changed both at time 250 ("posedege clk") and at time 305 ("negedge clk"). This is an unexpected behavior and we want to find the reason why these two conditions can not be true using only one instruction "ADD" with store address "rd=7".

(a) // bug occurs if RDwire != RDaddr5
RDwire = ((IR4[`op]==`SW) && (IR4[`rt]==5'd7) &&
(RDaddr5==5'd7) && (IR5[`op]==`ADD)) ?
5'd14 : RDaddr5;

(b) Diagnosis: DUV.IR2, suggested values to solve problem:
Variable DUV.IR2, at time 250 (posedge clk),
value= 32'b00000000000000000**00111**00000001001;
rd=7
Variable DUV.IR2, at time 305 (negedge clk, #5),
value= 32'b**100000**00000000000000000000000000;
ADD, illegal opcode
Variable DUV.IR2, at time 405 (negedge clk, #5),
value= 32'b**101011**00000**00111**0000000000000000;
SW rt=7

Fig. 5. Unreachability diagnosis example. At time 550, the suggested values will propagate to RDaddr5 with value 5'd7, IR4[`op] with value SW, and IR5[`op] with value ADD. The code that triggers the bug can then be reached.

We did further analysis and found that `ADD is illegal in the `op field, making the decoder produce 0 as the RD register address. As a result, RDaddr5 will never be 5'd7 when IR5[`op] is `ADD. This is why the bug can never be triggered. Without the diagnosis report, it took one engineer more than two hours to figure out what was wrong. With the report, the debugging time was reduced to just 10 minutes. This result suggests that our diagnosis techniques can provide valuable insights for debugging unreachability problems.

### B. Quantitative Evaluation

In this subsection we present the quantitative evaluation results of our diagnosis methods by showing the number of diag-

TABLE I
CHARACTERISTICS OF BENCHMARKS AND CAUSE OF CODE UNREACHABILITY. THE FIRST 6 CASES ARE DESIGN BUGS, WHILE THE LAST TWO ARE DUE TO OVER-CONSTRAINED TESTBENCHES.

| Case ID | Testbench configuration | Cause of unreachability |
|---|---|---|
| BUG20 | Properly-constrained random testbench, 14 cycles. | oldRS is always the same as RSaddr such that "(oldRS[4]==0) && (RSaddr[4]==1))" can never be true. |
| BUG22 | Properly-constrained random testbench, 14 cycles. | OPCODE can never be ADD with legal input pattern to make IR5[`op]==`ADD. |
| BUG29 | Properly-constrained random testbench, 14 cycles. | IR3[`op] can not be SW and one of ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI at the same time, which is a contradiction. |
| BUG31 | Properly-constrained random testbench, 14 cycles. | OPCODE can never be ADD with legal input pattern to make IR[`op]==`ADD. |
| BUG33 | Properly-constrained random testbench, 14 cycles. | branch_taken_reg and {IR[`op],IR[`function]}=={`SPECIAL,`SLL} can never be true at the same time. |
| BUG34 | Properly-constrained random testbench, 14 cycles. | branch_taken_reg and {IR[`op],IR[`function]}=={`SPECIAL,`SLL} can never be true at the same time. |
| CaseA | Constrained-random testbench: ADD, ADDI, SW, LW, NOP, 14 cycles. | Over-constrained input patterns. |
| CaseB | Constrained-random testbench: ADD, ADDI, SW, LW, SRL, SLL, SRA, BEQ, NOP, 14 cycles. | Over-constrained input patterns. |

noses compared with the total number of liberated variables in the design. In this way, we can evaluate how well our methods can narrow down the unreachability problem. To diagnose unreachability bugs in the DLX design, we prepared six properly-constrained testbenches. In order to show that our diagnosis techniques can also find errors in the testbench, we also developed two over-constrained testbenches that can only generate limited instructions. Since we performed formal reachability analysis for 14 cycles, in our diagnosis we also performed symbolic simulation for 14 cycles.

The results of unreachability diagnosis are shown in Table II. We first examined the effectiveness of our methods by doing full design diagnosis (the left half of the table). In other words, we assumed that any variable could be responsible for the problem and liberated all the variables in the design except the clock and the reset signals. The results show that even though we liberated all the variables in the design which consequently made the symbolic simulator explore all possible execution paths, the maximum runtime was still smaller than 30 minutes, suggesting that our method is efficient. In addition, the memory usage for most cases is less than 1 Gbyte, which can be executed on most modern computers. We also observed that the runtime and memory usage for different cases may vary because symbolic simulation needs to explore different paths due to different unreachability conditions.

In addition to runtime and memory usage, we also measured how well the causes of unreachability could be narrowed down by comparing the number of returned diagnoses and the number of liberated variables. Table II shows that the numbers of liberated variables for these cases are approximately 220[3]; however, the numbers of diagnoses are mostly smaller than 10. This result means that the user only needs to check approximate 4.5% of total variables to find where the problem is, showing that our methods can significantly narrow down the problem to a few key variables. It is important to note that BUG29 has a larger number of diagnoses because triggering the bug requires changing two liberated variables, while all other cases only require changing one variable. However, diagnosing such a case is still efficient in that runtime is still under 9 minutes. In our experience, diagnosing problems involving more than one liberated variable, such as BUG29, can be much more difficult than diagnosing problems that involve only one variable. In fact, one engineer thought that our diagnosis for BUG29 was incorrect because he did not think the design could have conflicts that the diagnosis pointed out. However, after carefully examining the results, it was found that our diagnosis was correct and the cause of unreachability is nontrivial. Without our diagnosis report, it would take the engineer much longer to figure out what was wrong.

To evaluate the relationship between the number of liberated variables and diagnosis efficiency, we also performed diagnosis on the buggy block by making only variables in that block liberated. The results are shown in the right half of Table II. It is observed that the numbers of liberated variables are substantially smaller than those on the left half; consequently, the execution time and memory consumption were greatly reduced. Moreover, the numbers of diagnoses were also smaller, indicating that if the user can provide more information such as erroneous blocks and the variables that do not need to be liberated, our diagnosis report will also become more precise. This can reduce the effort required to analyze the diagnosis results.

To check how well our technique can handle large designs which contain more liberated variables, we performed unreachability analysis on two industrial designs, and the results are shown in Table III. DesignA is a block in a multimedia system-on-chip circuit. We symbolically simulated 7 cycles for unreachability analysis. As shown in the Table III, 36 diagnoses that contained two liberated variables were reported. Note that when two liberated variables are considered simultaneously, there will be 27730 possible combinations of diagnoses, and it is highly unlikely to find the causes of unreachability without our unreachability analysis. However, with our technique, only 36 diagnoses needed to be checked, and approximately 99.8% possible combinations were eliminated. This result suggests that our techniques can greatly reduce an engineer's effort in analyzing unreachable code.

DesignB is a high-speed I/O interface which consists of many different types of modules. There are 33 instances in the design and 25 of them are instantiated from the same module under different hierarchy. Obviously, it is very difficult to manually analyze the cause of unreachability for such a complex structure. By applying our unreachability analysis for 50 cycles, we found that the diagnosis results saturated after 10

[3]The slight difference is due to registers or wires added for the manually-inserted bugs.

TABLE II
UNREACHABILITY DIAGNOSIS RESULTS. IN FULL DESIGN DIAGNOSIS, ALL THE VARIABLES IN THE DESIGN (INCLUDING TESTBENCH) ARE LIBERATED. IN BUGGY MODULE DIAGNOSIS, ONLY THE VARIABLES IN THE BUGGY MODULE ARE LIBERATED. DIAGNOSIS FOR BUG29 CONTAINS TWO LIBERATED VARIABLES, WHILE ALL OTHER DIAGNOSIS CONTAINS ONLY ONE VARIABLE.

| | | Full design diagnosis (including testbench) | | | | Buggy module diagnosis | | | |
|---|---|---|---|---|---|---|---|---|---|
| Cases | Buggy module | Run time | Memory (MB) | #Liberated Vars. | #Diagnoses | Run time | Memory (MB) | #Liberated Vars. | #Diagnoses |
| BUG20 | regfile.v | 3m20s | 637.36 | 217 | 1 | 45s | 357.73 | 10 | 1 |
| BUG22 | cpu.v | 1m22s | 504.95 | 216 | 7 | 43s | 417.61 | 67 | 3 |
| BUG29 | cpu.v | 8m49s | 601.92 | 217 | 24 | 4m52s | 549.41 | 68 | 9 |
| BUG31 | cpu.v | 14m45s | 815.19 | 216 | 5 | 8m1s | 655 | 67 | 2 |
| BUG33 | decode.v | 28m39s | 1146.15 | 218 | 1 | 13m26s | 584.22 | 16 | 1 |
| BUG34 | decode.v | 28m48s | 1146.35 | 218 | 1 | 13m24s | 584.33 | 16 | 1 |
| CaseA | tbench.v | 29m17s | 897.93 | 215 | 8 | 7m14s | 394.69 | 7 | 1 |
| CaseB | tbench.v | 29m13s | 887.22 | 215 | 8 | 7m31s | 394.29 | 7 | 1 |

cycles, and 15 diagnoses were reported. Note that the run time was still shorter than 33 minutes even for such a large design, suggesting that our technique provides a practical solution for finding the cause of unreachability in industrial-size designs.

TABLE III
UNREACHABILITY DIAGNOSIS RESULTS USING INDUSTRIAL DESIGNS. DESIGNA IS SIMULATED FOR 7 CYCLES, AND DESIGNB IS SIMULATED FOR 50 CYCLES.

| Cases | Lines of RTL | Run time | #Liberated Vars. | #Diagnosis |
|---|---|---|---|---|
| DesignA | 5074 | 1m34s | 236 | 36 |
| DesignB | 8068 | 32m5s | 1520 | 15 |

In our experiments we could always find valid diagnosis. If diagnosis could not be found, one should either simulate more cycles or increase the number of allowed liberated variables. The former case indicates insufficient verification depth, while the latter one suggests that the cause of the problem is sophisticated and solving it requires liberating more variables.

## VI. CONCLUSION

Code coverage is a key metric in hardware verification; however, diagnosing the root cause of code unreachability remains a challenging and tedious task due to the lack of effective diagnosis methodologies and tools. In this work, we performed an analytical study on the unreachability problem and proposed an unreachability diagnosis technique that can identify the cause of unreachability. Moreover, our technique can provide a set of suggestions on how to fix the unreachability problem. This is achieved by our enhanced symbolic simulation algorithm that can search unexplored execution paths, as well as our new method that uses symbolic conditions to produce diagnosis reports. Our empirical results using DLX and two industrial designs show that the proposed diagnosis technique can significantly narrow down the causes of the unreachability problems: the user only needs to check approximately 4.5% of total variables to find where the problem is. These results suggest that our unreachability diagnosis technique can significantly reduce the debugging time of unreachability problems.

## ACKNOWLEDGMENT

This research was supported by the National Science Council, Taiwan under Grant NSC 97-2221-E-002-216-MY3.

## REFERENCES

[1] M. F. Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith and M. Abadir, "Debugging Sequential Circuits Using Boolean Satisfiability," *ICCAD'04*, pp. 204-209.

[2] M. F. Ali, S. Safarpour, A. Veneris, M. S. Abadir and R. Drechsler, "Post-Verification Debugging of Hierarchical Designs," *ICCAD'05*, pp. 871-876.

[3] V. Boppana, I. Ghosh, R. Mukherjee, J. Jain, and M. Fujita "Hierarchical Error Diagnosis Targeting RTL Circuits," *VLSI design'00*, pp. 436-441.

[4] D. Brand, R. A. Bergamaschi, and L. Stok, "Be Careful with Don't Cares," *ICCAD'95*, pp.83-86.

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

[6] K. H. Chang, I. Wagner, V. Bertacco, and I. L. Markov, "Automatic Error Diagnosis and Correction for RTL Designs," *HLDVT'07*, pp. 65-72.

[7] 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.

[8] 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.

[9] G. Cunningham, B. Jackson, and J. Dines, "Expression Coverability Analysis: Improving Code Coverage with Model Checking," *DVCON'04*.

[10] A. Datta and V. Singhal, "Formal Verification of a Public-Domain DDR2 Controller Design," *VLSI design'08*, pp. 475-480.

[11] N. Éen and N. Sörensson, "An Extensible SAT-solver," *SAT'03*, pp. 502-518.

[12] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic Fault Localization for Property Checking," *IEEE TCAD*, 2008, vol. 27(6), pp. 1138-1149.

[13] J. L. Hennessy, and D. J. Patterson, "Computer Architecture: A Quantitative Approach, 2nd edition," Morgan Kaufman, 1996.

[14] M. Janota, R. Grigore, and M. Moskal, "Reachability Analysis for Annotated Code," *SAVCBS'07*, pp. 23-30.

[15] T. Y. Jiang, C. N. Liu, and J. Y. Jou, "Effective Error Diagnosis for RTL Designs in HDLs," *ATS'02*, pp. 362-367.

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

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

[18] S. Y. Kuo, "Locating Logic Design Errors via Test Generation and Don't-Care Propagation," *EDAC'92*, pp 466-471.

[19] J. C. Madre, O. Coudert and J. P. Billon, "Automating the Diagnosis and the Rectification of Design Errors with PRIAM," *ICCAD'89*, pp 30-33.

[20] C. H. Shi and J. Y. Jou, "An Efficient Approach for Error Diagnosis in HDL Design," *ISCAS'03*, pp. 732-735.

[21] A. Smith, A. Veneris and A. Viglas, "Design Diagnosis Using Boolean Satisfiability," *ASPDAC'04*, pp. 218-223.

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

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