Automatic Error Diagnosis and Correction for RTL Designs

Kai-hui Chang, Ilya Wagner, Valeria Bertacco, Igor L. Markov
EECS Department, University of Michigan, Ann Arbor, MI 48109-2121
{changkh, iwagner, valeria, imarkov}@umich.edu

ABSTRACT

Recent improvements in design verification strive to automate the error-detection process and greatly enhance engineers' ability to detect functional errors. However, the process of diagnosing the cause of these errors and fixing them remains difficult and requires significant ad-hoc manual effort. Our work proposes improvements to this aspect of verification by presenting novel constructs and algorithms to automate the error-repair process at the Register-Transfer Level (RTL), where most development occurs. Our contributions include a new RTL error model and scalable error-repair algorithms. Empirical results show that our solution can diagnose and correct errors in just a handful of minutes even for complex designs of up to several thousand lines of RTL code. This demonstrates the superior scalability and efficiency of our approach compared to previous work.

1. INTRODUCTION

The dramatic increase in design complexity of modern electronics challenges our ability to ensure its functional correctness. While improvements in verification allow engineers to find a larger fraction of design errors more efficiently, little effort has been devoted to fixing such errors. As a result, debugging remains an expensive and challenging task. To address this problem, researchers have proposed techniques that automate the debugging process, by locating the error source within a design and/or by suggesting possible corrections. Although these techniques are successful to some extent, they mainly focus on the gate level [7, 15, 21, 22, 23] or the transistor level [14]. However, most debugging effort occurs in the Register-Transfer Level (RTL) description of a circuit, where design activities take place. The lack of powerful and automatic debugging tools at this level greatly reduces designers' productivity when fixing even very simple errors. Leveraging gate-level diagnosis tools for the RTL, however, is difficult because synthesis tools blur the mapping between the RTL code and the gate-level netlist.

To address this problem, techniques that work directly at the RTL have been developed recently. The first group of techniques [11, 16, 18] employs a software analysis approach that implicitly uses multiplexers (MUXes) to identify statements in the RTL code that are responsible for the errors. However, these techniques can return large potential error sites. To address this problem, a recent work by Staber *et al.* [20] explicitly inserts MUXes into the HDL code. This approach allows the use of hardware analysis techniques and greatly improves the accuracy of error diagnosis. The third group of techniques, such as [6], analyzes an HDL description and failed properties using state-transition diagram and model checking; because of that these techniques are more suitable for formal verification flows, rather than the simulation-based flows prevalent in industry. In addition, all three groups of techniques cannot repair identified errors automatically. Finally, the work in [19] can diagnose and correct RTL design errors automatically, but it relies on state-transition analysis and hence, it does not scale beyond tens of state bits. Furthermore, this algorithm requires a correct formal specification of the design, which is rarely available in today's design environments, because its development is often as challenging as the design process itself. In contrast, the most common type of specification is a high-level model, often written in a high-level language, which produces the correct I/O behavior of the system.

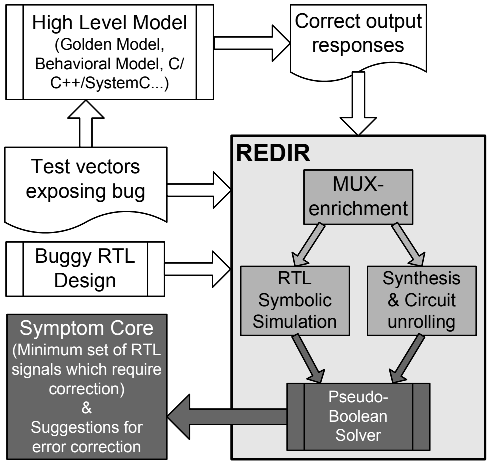
High Level Model
(Golden Model, Behavioral Model, C/ C++/SystemC...)
Correct output responses
REDIR
Test vectors exposing bug
MUX-enrichment
Buggy RTL Design
RTL Symbolic Simulation
Synthesis & Circuit unrolling
Symptom Core
(Minimum set of RTL signals which require correction)
&
Suggestions for error correction
Pseudo-Boolean Solver

Figure 1: REDIR framework. Inputs to the tool are an RTL design (which includes one or more errors), test vectors exposing the bug(s), and correct output responses for those vectors obtained from a high-level simulation. Outputs of the tool include REDIR *symptom core* (a minimum cardinality set of RTL signals which need to be modified in order to correct the design), as well as suggestions to fix the errors.

To develop a scalable and powerful RTL error diagnosis and correction system, we adopt hardware analysis techniques that are prevalent at the gate-level into the RTL. This approach is significantly more accurate than previous software-based solutions in that we can analyze designs rigorously using formal hardware verification techniques. At the same time, it is considerably faster and more scalable than gate-level diagnosis because errors are modeled at a higher level. Similar to several successful gate-level methods [2, 3, 7, 21, 23], it only requires test vectors and output responses, making it more practical than existing formal analysis solutions. Finally, the novel error model and increased accuracy of our approach allow our technique to provide insightful suggestions for correcting diagnosed errors. Our main contributions include: (1) a new RTL error model that explicitly inserts MUXes into RTL code to model errors in the design's internal signals, as opposed to previous solutions that use MUXes to model errors in RTL code statements; (2) innovative error-diagnosis algorithms using synthesis or symbolic simulation; and (3) a novel error-correction technique based on *signal signatures* (value samples) that is especially suitable for the RTL. Our empirical results show that these techniques allow us to provide highly accurate diagnoses very quickly.

We implemented our solution in a framework called *REDIR* (RTL Error DIagnosis and Repair), highlighted in the diagnosis and correction flow of Figure 1. The inputs to the framework include a design containing one or more bugs, a set of simulation vectors exposing them, and the correct responses for the primary outputs over the given test vectors (usually generated by a high-level behavioral model written in C, C++, SystemC, etc). Note that we only require the correct responses at the primary outputs of the high-level model and no internal values are required. The correct output responses could be the primary outputs of the design, or the outputs of a set of checkers in the context of assertion-based verification. REDIR can then output a minimum cardinality set of RTL signals that should be corrected in order to eliminate the erroneous behavior. We call this set the *symptom core*. When multiple cores exist, REDIR provides all of the possible minimal cardinality sets. In addition, the framework suggests several possible ways of modifying the signals in the symptom core to help in the correction of the design. Our empirical evaluation shows that REDIR can diagnose and correct multiple errors in design descriptions with thousands of lines of Verilog code (or approximately 100K cells after synthesis), which is approximately the block size that a single engineer actively works on. As a result, REDIR can assist in everyday debugging tasks and fundamentally accelerate the design development.

The rest of the paper is organized as follows. In Section 2, we describe the related work and provide the necessary background. Section 3 and Section 4 describe our error diagnosis and correction techniques, respectively. Empirical results are given in Section 5, while Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

Our error-diagnosis algorithm converts the error-diagnosis problem into a *Pseudo-Boolean (PB)* problem, and then uses a PB solver to perform the diagnosis and infer which design signals are responsible for incorrect output behavior. In this section, we first define PB problems, which are an extension of *SATisifiability (SAT)* problems. Next, we review gate-level diagnosis techniques, which provide the foundation for our synthesis-based diagnosis, and are used for comparison in our experimental results. Finally, we show the basic idea behind symbolic simulation, which we use as an alternative, compact way to formulate the PB problem.

2.1 Pseudo-Boolean Problems

PB problems, also called 0-1 integer linear programming problems, are an extension of SAT problems. PB constraints are specified as an inequality with a linear combination of Boolean variables: $C_0 p_o + C_1 p_1 + ... + C_{n-1} p_{n-1} \geq C_n$, where the variables p_i are defined over the Boolean set $\{0, 1\}$. A PB problem allows the use of an additional *objective function*, which is a linear expression that should be minimized or maximized under the given constraints. A number of PB solvers have been developed recently by extending existing SAT solvers (for instance, MiniSat+ [10]).

2.2 Gate-Level Error Diagnosis Techniques

Gate-level error diagnosis techniques have been studied extensively in the past. Early work often relies on specific error models to simplify the problem, such as [1, 15]. Recently, the power and effectiveness of gate-level error diagnosis have been improved by the work of Smith *et al.* [21], which does not rely on any error model. In Smith's technique, two types of components are added to a given buggy netlist. These components include (1) multiplexers, and (2) an error-cardinality constraint. The purpose of the multiplexers is to model errors – when their select lines are asserted, alternative sources drive the corresponding internal wires to correct the output responses. The number of asserted select lines is limited by the error-cardinality constraint, which is implemented as an adder and a comparator: the adder counts the number of asserted select lines, and its output is forced to a value N using the comparator. The circuit is then converted into Conjunctive Normal Form (CNF), and inputs and outputs are subjected to additional constraints from input vectors and correct output responses, obtained from a high-level model. Error diagnosis is then performed by iteratively solving the CNF using a SAT solver with an increasing value for N, until a solution is found.

Note that Smith's technique diagnoses errors in combinational circuits only; to diagnose sequential circuits, Ali *et al.* [3] extended Smith's work by unrolling the circuit M times before the CNF conversion step, where M is the sequential length of the given trace. REDIR uses a similar approach. Since the trace may be extremely long, we perform trace minimization in REDIR before error diagnosis, and this step makes REDIR more practical for real designs. We also show in Section 5 that our algorithm runs significantly faster and is more accurate than Ali's techniques, since we model errors at the RTL instead of the gate level.

2.3 Logic vs. Symbolic Simulation

Logic simulation models the behavior of a digital circuit by propagating scalar Boolean values (0 and 1) from primary inputs to primary outputs. For example, when simulating 2-input AND with both inputs set to 1, the output 1 is produced. On the other hand, *symbolic simulation* uses symbols instead of scalar values and produces Boolean expressions at the outputs [4, 5]. As a result, simulating a 2-input XOR with inputs *a* and *b* generates an expression "*a* XOR *b*" instead of a scalar value. To improve scalability, modern symbolic simulators employ several techniques, including approximation, parameterization and on-the-fly logic simplification. For example, with on-the-fly logic simplification, "*0* XOR *b*" is simplified to *b* thus reducing the complexity of the expression. Traditional symbolic simulators operate on a gate-level model of a design; however, in recent years simulators operating on RTL descriptions have been proposed [12, 13]. Symbolic simulation is an alternative way to generate an instance of the Pseudo-Boolean constraint problem that we use in our error diagnosis framework.

3. RTL ERROR DIAGNOSIS

In this section, we describe our error-diagnosis techniques. First, we explain our RTL error model, and then propose two diagnosis methods that use either synthesis (Section 3.2) or symbolic simulation (Section 3.3). Finally, we outline how hierarchical designs should be handled.

3.1 Error Modeling

In our framework the error-diagnosis problem is represented with (1) an RTL description containing one or more bugs that is composed of variables (wires, registers, I/O) and operations on those variables; (2) a set of test vectors exposing the bugs; and (3) the correct output responses for the given test vectors. The objective of error diagnosis is to identify a minimal number of variables in the RTL description that are responsible for the design's erroneous behavior. Moreover, errors can be corrected by modifying the statements related to those variables. Each signal affecting the design's correctness is called a *symptom variable*. Without minimization, the set of symptom variables reported would include the root cause of the bug and also all the signals in the cone of logic emanating from it: correcting all the symptom variables on any cut across this cone of logic would eliminate the bug. Therefore, by forcing the PB solver to minimize the number of symptom variables, we return a set of signals (or symptom core) as close to the root cause of the erroneous behavior as possible.

To model errors in a design, we introduce a conditional assignment for each RTL variable, as shown in the example in Figure

2. Note that we insert only one conditional assignment even if the variable contains multiple bits. These assignments allow the REDIR framework to locate sites of erroneous behavior in RTL, as we illustrate using a *half_adder* design shown in Figure 2. Suppose that the output responses of the design are incorrect because c should be driven by "a & b" instead of "$a \mid b$". Obviously, to produce the correct outputs, the behavior of c must be changed. To model this situation, we insert a conditional assignment, "assign c_n = c_{sel} ? c_f : c", into the code. Next, we replace all occurrences of c in the code with c_n, except when c is used on the left-hand-side of an assignment. We call c_{sel} a *select variable* and c_f a *free variable* in this paper. Then, by asserting c_{sel} and using an alternative signal source, modeled by c_f, we can force the circuit to behave as desired. If we can identify the select variables that should be asserted and the correct signals that should drive the corresponding free variables to produce correct circuit behavior, we can diagnose and fix the errors in the design.

```
module half_adder(a, b, s, c);
  input a, b; output s, c;
  assign s = a ^ b;
  assign c = a | b;
endmodule
module half_adder_MUX_enriched(a, b, s_n, c_n,
s_sel, c_sel, s_f, c_f);
  input a, b, s_sel, c_sel, s_f, c_f;
  output s_n, c_n;
  assign s = a ^ b;
  assign c = a | b;
  assign s_n = s_sel ?  s_f :  s;
  assign c_n = c_sel ?  c_f :  c;
endmodule
```

Figure 2: An RTL error-modeling code example: module half_adder shows the original code, where c is erroneously driven by "$a \mid b$" instead of "a & b"; and module half_adder_MUX_enriched shows the MUX-enriched version. The differences are marked in boldface.

The procedure to introduce a conditional assignment for a design variable v is called MUX-enrichment (since conditional assignments are conceptually multiplexers), and its pseudo-code is shown in Figure 3. It should be performed on each internal signal, defined in the circuit, including registers. The primary inputs, however, should not be MUX-enriched since by construction they cannot have erroneous values. It also should be noted that for hierarchical designs the primary inputs of a module may be driven by the outputs of another module and, therefore, may assume erroneous values. To handle this situation, we insert conditional assignments into the hierarchical modules' output ports.

```
procedure MUX_enrichment($v$)
  create a new signal wire $v_n$ and new inputs $v_f$ and $v_{sel}$;
  add conditional assignment "$v_n$ = $v_{sel}$ ? $v_f$ : $v$";
  replace all occurrences of $v$ that appear on the right-hand-
    side of assignments (including outputs, if/case conditions,
    etc.) with $v_n$;
```

Figure 3: Procedure to insert a conditional assignment for a signal in an RTL description for error-modeling.

Example 1. Using the half_adder example shown in Figure 2. A test vector that exposes the design error assigns 1 to a and 0 to b. By simulation, s is 1 and c is 1. However, the correct output responses for c should be 0 instead of 1. In other words, c_n should be 0. To satisfy the conditional assignments that we inserted during MUX-enrichment, c_{sel} is forced to 1 and c_f must be 0. That c_{sel} is asserted means signal c is erroneous, which provides the correct diagnosis. Techniques to determine the select variables that should be asserted will be described in the following two sections.

3.2 Diagnosis with Synthesis

After the error-modeling constructs are inserted into a design, error diagnosis identifies the minimal number of select variables that should be asserted, along with the values of their corresponding free variables, to generate the correct output responses. In this section we present an error-diagnosis technique that uses synthesis and circuit unrolling. In contrast with existing gate-level diagnosis techniques described in Section 2.2, our RTL error-modeling constructs are synthesized with the design, which eliminates the need to insert multiplexers at the gate level. Thus, the synthesized netlist faithfully preserves the constructs inserted at the RTL, enabling accurate RTL error diagnosis. This is significantly different from diagnosing errors at the gate level, since synthesis is only used to generate Boolean expressions between RTL variables, and the synthesized netlist is not the target of the diagnosis. As a result, our diagnosis method has a much smaller search space and runs significantly faster than gate-level techniques, as we show in our experimental results.

```
Procedure syn_based_diagnosis(designCNF, c, inputs, outputs);
  CNF = unroll designCNF c times;
  connect all select variables in CNF to those in the first cycle;
  constrain PI/PO in CNF using inputs/outputs;
  PBC = CNF, min( $\sum$ select variables);
  return solution= PB-Solve(BPC);
```

Figure 4: Procedure to perform error diagnosis using synthesis and circuit unrolling.

Figure 4 outlines the algorithm for synthesis-based error diagnosis. Before the procedure is called, the design is synthesized and its combinational portion is converted to CNF format (*designCNF*). Other inputs to the procedure include the length of the bug trace, c, as well as the test vectors (*inputs*) and their correct output responses (*outputs*). To make sure that the diagnosis applies to all simulation cycles, the algorithm connects the select variables for each unrolled copy to the corresponding CNF variables in the first copy. On the other hand, free variables for each unrolled copy of the circuit are independent. When a solution is found, each asserted select variable is a symptom variable, and the solution for its corresponding free variable is an alternative signal source that can fix the design errors. Note that if state values over time are known, they can be used to constrain the CNF at register boundaries, reducing the sequential error-diagnosis problem to combinational. The constructed CNF, along with the objective to minimize the sum of select variables, forms a *Pseudo-Boolean Constraint (PBC)*. Error diagnosis is then performed by solving the PBC.

3.3 Diagnosis with RTL Symbolic Simulation

In this section we propose an alternative error-diagnosis technique that potentially scales further than the synthesis-based technique. This is achieved by performing symbolic simulation directly on the RTL representation and generating Boolean expressions at the primary outputs for all simulated cycles. The outputs' Boolean expressions are used to build a PB problem's instance, that is then handed over to a PB solver for error diagnosis.

Although RTL symbolic simulators are not yet commonly available in the industry, effective solutions have been proposed in recent years in the literature [12, 13]. Moreover, because of the scalability advantages of performing symbolic simulation at the RTL instead of the gate level, commercial-quality solutions are starting to appear. For our empirical validation we used one such experimental RTL symbolic simulator [24].

Figure 5 illustrates our novel procedure that uses symbolic simulation and PB solving. We assume that the registers are initialized to known values before the procedure is invoked. We also assume that the circuit contains n MUX-enriched signals named v_i, where $i = \{1..n\}$. Each v_i has a corresponding select variable v_{i_sel} and a free variable v_{i_f}. There are o primary outputs, named PO_j, where $j = \{1..o\}$. We use subscript "@" to prefix the cycle during which the symbols are generated. For each primary output j and for each cycle t we compute expression $PO_{j@t}$ by symbolically simulating the given RTL design, and also obtain correct output value $CPO_{j@t}$ from the high-level model. The inputs to the procedure are the RTL design (*design*), the test vectors (*test_vectors*), and the correct output responses over time (*CPO*).

```
Procedure $sim\_based\_diagnosis(design, test\_vectors, CPO)$;
1  $\forall i, 1 \leq i \leq n$, $\mathbf{v}_{i\_sel} = new\_symbol()$;
2  for $t = 1$ to $c$ begin // Simulate $c$ cycles
3      PI = $test\_vector$ at cycle $t$;
4      $\forall i, 1 \leq i \leq n$, $\mathbf{v}_{i\_f@t} = new\_symbol()$;
5      $PO_{@t} = simulate(design)$;
6  end
7  $PBC = \bigwedge_{j=1}^{o} \bigwedge_{t=1}^{c} (\mathbf{PO}_{j@t} = CPO_{j@t})$, $\min(\sum_{i=1}^{n} \mathbf{v}_{i\_sel})$;
8  return solution= $PB\_Solve(PBC)$;
```

Figure 5: Procedure to perform error diagnosis using symbolic simulation. The boldfaced variables are symbolic variables or expressions, while all others are scalar values.

In the algorithm shown in Figure 5, a symbol is initially created for each select variable (line 1). During the simulation, a new symbol is created for each free variable in every cycle, and test vectors are applied to primary inputs, as shown in lines 2-4. The reason for creating only one symbol for each select variable is that a conditional assignment should be either activated or inactivated throughout the entire simulation, while each free variable requires a new symbol at every cycle because the value of the variable may change. As a result, the symbols for the select variables are assigned outside the simulation loop, while the symbols for the free variables are assigned in the loop. The values of the free variables can be used as the alternative signal source to produce the correct behavior of the circuit. After simulating one cycle, a Boolean expression for all of the primary outputs are created and saved in $PO_{@t}$ (line 5). After the simulation completes, the generated Boolean expressions for all the primary outputs are constrained by their respective correct output values and are ANDed to form a *PBC* problem as line 7 shows. In order to minimize the number of symptom variables, we minimize the *sum* of select variables, which is also added to the *PBC* as the objective function. A PB solver is then invoked to solve the formulated *PBC*, as shown in line 8. In the solution, the asserted select variables represent the symptom variables, and the values of the free variables represent the alternative signal sources that can be used to correct the erroneous output responses.

3.4 Handling Hierarchical Designs

Current designs often have hierarchical structures to allow the circuit to be decomposed into smaller blocks and thus reduce its complexity. In this subsection we discuss how the MUX-enriched circuit should be instantiated if it is encapsulated as a module in such a hierarchical design.

The algorithm to insert MUXes into a single module m is shown in Figure 3. If m is instantiated inside of another module M, however, MUX-enrichment of M must include an extra step where new inputs are added to all instantiations of m. Therefore, for hierarchical designs, the insertion of conditional assignments must be performed bottom-up: MUX-enrichment in a module must be executed before it is instantiated by another module. This is achieved by analyzing the design hierarchy and performing MUX-enrichment in a reverse-topological order. It is important to note that in hierarchical designs, select variables from instances of the same module should be shared, while free variables should not. This is because instances of the same module have the same symptom variables. As a result, select variables should share the same signals. On the other hand, each instance is allowed to have different values for their internal signals; therefore, each free variable should have its own signal. However, a bug may require fixing only one RTL instance while other instances can be left intact. This problem can be solved by providing separate select variables for different instances.

4. RTL ERROR CORRECTION

The RTL error-correction problem is formulated as follows: given an erroneous RTL description, find a variant description for one or more of the modules that compose it so that the new design produces correct output responses for the entire set of given bug traces. Although many error-repair techniques exist for gate-level designs, very few studies focus on RTL. One major reason is the lack of logic representations that can support the manipulations required for error correction, especially in hierarchical designs. For example, a variable in an RTL module that is instantiated multiple times will implement many different functions depending on where it is instantiated. Solving for all the functions simultaneously to find a correct fix is not an easy task. In [7] Chang *et al.* proposed a solution for gate-level error correction using *signatures,* a collection of a wire's simulation values represented by a bit vector. Since signatures can be easily calculated via simulation, this technique is especially suitable for RTL. To support the error-correction requirements of RTL, we propose a new scheme based on similar concepts. In this section, we first describe a baseline error-correction technique and then show how we extend it to hierarchical and sequential designs. Finally, we provide insights obtained on the implementation of our system.

4.1 Baseline Error Correction Technique

Our baseline correction technique considers a flat combinational design, where all internal signatures and a symptom core are available, and applies the following steps: (1) the signatures of the symptom cores are replaced by the correct values of the corresponding free variables; (2) we generate partial truth-tables based on these correct signature values – terms not appearing in the tables are don't-cares; (3) we synthesize the truth-tables, produce new logic expressions for the symptom variables, and then use them to replace the original erroneous expression.

Example 2. Using the same circuit as Example 1. Two test vectors are applied: (a,b)= (0, 1) and (1, 1). According to the correction procedure, the signatures for a, b, s and c are 10, 11, 01, 11, respectively (the least-significant bits are the simulation values of the first vector). Then error diagnosis asserts c_{sel}; hence, the symptom core contains only one variable, c. The signature of c is replaced by the values returned in c_f for the two applied test vectors and becomes 10. We then build the following truth-table (note that the bits in the signatures are written vertically), where the missing minterms are all don't-cares:

a	b	c
1	1	1
0	1	0

Finally, the synthesis step may produce several equations, including: "c= a", "c=a&b". Note that due to the partial information given by the bug trace, several fixes are provided by REDIR. In general longer traces can lead to more accurate corrections because they provide more information.

Benchmark	Description	#Cells	#FFs	Trace type	#Lines	#Assign
Pipe	Part of PicoJava pipeline control unit	55	2	Constrained-random	264	31
Pre_norm	Part of FPU	1877	71	Constrained-random	270	43
MD5	MD5 full chip	13311	910	Direct test	438	37
MiniRISC	MiniRISC full chip	6402	887	Direct test	2013	43
CF_FFT	Part of the CF_FFT chip	126532	16638	Constrained-random	998	223
DLX	5-stage pipeline CPU running MIPS-Lite ISA	14725	2062	Constrained-random	1225	84
Alpha	5-stage pipeline CPU running Alpha ISA	38299	2917	Constrained-random	1841	134

Table 1: Characteristics of benchmarks. "#Cells" is the cell count of the synthesized netlist, "#FFs" is the number of flip-flops, "#Lines" is the number of lines of RTL code in a design, and "#Assign" is the number of inserted conditional assignments.

4.2 Hierarchical and Sequential Designs

In a flattened design, each RTL variable represents exactly one logic function. In a hierarchical design, however, each variable may represent more than one logic function. Therefore, we devise the following techniques to construct the signatures of RTL variables. For clarity, we call a variable in an RTL module a *module variable* and a variable in an instance generated by the module an *instance variable*. A module variable may generate multiple instance variables if the module is instantiated several times.

In RTL error correction, we modify the source code of the modules to correct the design's behavior. Since changing an RTL module affects all the instances produced by the module, we concatenate the simulation values of the instance variables derived from the same module variable to produce a signature for the module variable. Therefore, we can guarantee that a change in a module will affect all instances of this module in the same way. Similarly, we concatenate the signatures of the module variable at different cycles for sequential error correction. A signature-construction example is given in Figure 6. Note that to ensure the correctness of error repair, the same instance and cycle orders must be used during the concatenation of signatures for all module variables.

Design:

```
  module top;
    half_adder c1(), c2();
  endmodule
```

Simulation values:

Cycle 0: top.c1.a = 0, top.c2.a = 0, top.c1.b = 1, top.c2.b = 0
Cycle 1: top.c1.a = 1, top.c2.a = 0, top.c1.b = 1, top.c2.b = 1

Constructed signature for RTL error correction:

$$\text{half\_adder.a} = \underbrace{\overbrace{1}^{c1.a}\;\overbrace{0}^{c2.a}}_{cycle\ 1}\underbrace{\overbrace{0}^{c1.a}\;\overbrace{0}^{c2.a}}_{cycle\ 0} \qquad \text{half\_adder.b} = \underbrace{\overbrace{1}^{c1.b}\;\overbrace{1}^{c2.b}}_{cycle\ 1}\underbrace{\overbrace{1}^{c1.b}\;\overbrace{0}^{c2.b}}_{cycle\ 0}$$

Figure 6: Signature-construction example. Simulation values of variables created from the same RTL variable at all cycles should be concatenated for error correction.

4.3 Identifying Erroneous Code Statements

Several existing error-diagnosis techniques are able to identify the RTL code statements that may be responsible for the design errors [11, 16, 18, 20]. Unlike these techniques, REDIR returns the RTL variables that are responsible for the errors instead. Since one variable may be affected by multiple statements, the search space of the errors modeled by these techniques tend to be larger than REDIR, making REDIR more efficient in error diagnosis. On the other hand, being able to identify erroneous statements may further localize the errors and make debugging easier. To achieve this goal, we observe that in correctly designed RTL code, the value of a variable should be affected by at most one statement at each cycle. Otherwise, a multiple-driven error will exist in the code. Based on this observation, we design the following procedure to identify the erroneous code statements using our error-diagnosis results.

Given a symptom variable, we first record the cycles at which the values of its free variables are different from its simulated values. Next, we identify the statements that assign new values to the symptom variable for those cycles: these statements are responsible for the errors. Since many modern logic simulators provide the capability to identify executed statements (e.g., full-trace mode in Cadence Verilog-XL), erroneous statements can be pinpointed easily by replaying the same traces used for diagnosis. After erroneous statements are identified, signatures for error-correction can be generated using only the cycles when the statements are executed. In this way, we can produce corrections specifically for the erroneous statements.

4.4 Implementation Insights

Currently, we only use one conditional assignment for a variable even if it is a bit vector. Often all the bits in a bit vector undergo the same manipulation, hence this simplification greatly reduces the search space for error diagnosis. On the other hand, we also provide the option to require a conditional assignment for each bit, if distinct bits in a vector take different roles. Note also that REDIR corrects a design only with respect to the traces provided, and may corrupt the responses of other test vectors not provided for the diagnosis. Thus, once repaired, a design should still undergo verification. If verification fails, new bug traces should also be diagnosed to provide more accurate fixes.

5. EXPERIMENTAL RESULTS

In our experiments, we evaluated the performance of the techniques described in this paper with a range of Verilog benchmarks. We used a proprietary Perl-based Verilog parser to insert conditional assignments into RTL code. Synthesis-based diagnosis was implemented using OpenAccess 2.2 and OAGear 0.96 [27] with RTL Compiler v4.10 from Cadence as synthesis tool. For simulation-based diagnosis, we adopted an experimental RTL symbolic simulator, Insight 1.4, from Avery Design Systems [24]. For efficiency, we implemented the techniques described in [10] to convert PB problems to SAT problems and adopted MiniSat as our SAT solver [9]. All the experiments were conducted on an AMD Opteron 880 (2.4GHz) Linux workstation with 16GB memory. Our benchmarks included several circuits selected from OpenCores [26] (Pre_norm, MD5, MiniRISC, and CF_FFT), the picoJava-II microprocessor (Pipe), DLX, and Alpha. The characteristics of the benchmarks are summarized in Table 1. Bugs (described in Table 2) were injected into these benchmarks, with the exception of DLX and Alpha, which already included bugs [25]. We used constrained-random simulation to generate bug traces for Pipe, Pre_norm, and CF_FFT, while we used the verification environment shipped with the designs for other benchmarks. Traces to expose bugs in DLX and Alpha were provided by the engineering team, who used a constrained-random simulation tool.

5.1 Synthesis-based Error Diagnosis

In our first experiment, we performed combinational and sequential error diagnosis using the synthesis-based techniques described in Section 3.2, and the results are summarized in Table 3. Recall

Benchmark	Bug ID	#Traces	#Cycles	Combinational			Sequential		
				Errors found		Runtime (sec)	Errors found		Runtime (sec)
				#Symp.	#Cores		#Symp.	#Cores.	
Pipe	A	32	200	1	1	6.0	1	1	**6.0**
Pre_norm	A	32	20	1	1	13.2	1	1	**13.2**
	B	32	20	1	1	11.4	1	2	**13.4**
	C	32	20	1	1	11.4	1	1	**11.4**
	D	32	20	2	1	12.4	2	2	**13.8**
	E	32	20	3	2	13.9	3	4	**17.4**
MD5	A	1	200	1	1	83.3	1	3	**173.2**
	B	1	200	1	1	42.9	1	2	**110.1**
	C	1	50	1	1	14.1	1	6	**49.8**
MRISC	A	1	500	States unavailable			1	2	**32.0**
CF_FFT	A	32	15	1	4	364.8	Trace unavailable		
DLX	A	1	150	1	1	41.2	1	3	**220.8**
	B	1	68 (178)	1	4	54.8	1	17	**1886.3**
	C	1	47 (142)	1	5	15.8	1	11	**104.0**
	D	1	77 (798)	1	3	27.5	1	9	**2765.1**
	E	1	49 (143)	1	4	19.1	1	12	**105.2**
	F	1	188	1	2	67.8	1	2	**457.4**
	G	1	30 (1080)	1	1	11.3	Trace unavailable		
Alpha	A	1	70(256)	1	5	127.4	1	9	**525.3**
	B	1	83(1433)	1	5	111.6	1	5	**368.9**
	C	1	150(9950)	1	3	122.3	1	3	**250.5**

Table 3: Synthesis-based error-diagnosis results. "#Symp." is the number of symptom variables in each core, and '#Cores" is the total number of symptom cores. Bug traces for several DLX/Alpha benchmarks have been minimized before diagnosis, and their original lengths are shown in parentheses.

Bench-mark	Bug ID	Description
Pipe	A	One signal inverted
Pre_norm	A	Reduced OR replaced by reduced AND
	B	One signal inverted
	C	One 26-bit bus MUX select line inverted
	D	Bug A + Bug B
	E	Bug A + Bug B + Bug C
MD5	A	Incorrect operand for a 32-bit addition
	B	Incorrect state transition
	C	Bug B with a shorter trace
MRISC	A	Incorrect RHS for a 11-bit value assignment
CF_FFT	A	One signal inverted
DLX	A	SLL inst. does shift the wrong way
	B	SLTIU inst. selects the wrong ALU operation
	C	JAL inst. leads to incorrect bypass from MEM stage
	D	Incorrect forwarding for ALU+IMM inst.
	E	Does not write to reg31
	F	RT reads lower 30 bits only
	G	If RT = 7 memory write is incorrect
Alpha	A	Write to zero-reg succeeds if rdb_idx = 5
	B	Forwarding through zero reg on rb
	C	Squash if source of MEM/WB = dest. of ID/EX and instr. in ID is not a branch

Table 2: Description of benchmark bugs. DLX and Alpha include native bugs, while the others were manually injected.

that a symptom core suggests a set of signals to be modified to correct the design, and it includes one or more symptom variables. In all our experiments, we found that the reported symptom cores included the root causes of errors for all the benchmarks. In other words, REDIR accurately pointed out the signals that exhibited incorrect behavior.

Comparison between combinational and sequential diagnosis: the difference between combinational and sequential diagnosis is that sequential diagnosis only uses output responses for constraints, while combinational is allowed to use state values. As Table 3 shows, the runtime of combinational diagnosis is typically shorter, and the number of symptom cores is often smaller. In DLX(D), for example, the combinational technique runs significantly faster than sequential, and returns only three cores, while sequential returns nine. The reason is that combinational diagnosis allows the use of state values, which provide additional constraints to the PB instance. As a result, the PB solver can find solutions faster, and the additional constraints further localize the bugs. Being able to utilize state values is especially important for designs with very deep pipelines, where an error may be observed hundred cycles later. For example, the error injected into CF_FFT requires more than 40 cycles to propagate to any primary output, making the use of sequential diagnosis difficult. In addition, bugs that are observed in design states can only be diagnosed when state values are available, such as DLX(G). On the other hand, sequential diagnosis is important when state values are unavailable. For example, the bug injected into the MiniRISC processor changed the state registers, damaging correct state values. In practice, it is also common that only responses at primary outputs are known. Therefore, being able to diagnose errors in combinational and sequential circuits is equally important, and both are supported by REDIR.

The comparison between MD5(B) and MD5(C) shows that there is a trade-off between diagnosis runtime and quality: MD5(C) uses a shorter trace and thus requires shorter diagnosis runtime; however, the number of symptom cores is larger than that returned by MD5(B), showing that the results are less accurate. The reason is that longer traces usually contain more information; therefore, they can better localize design errors. One way to obtain short yet high-quality traces is to perform bug trace minimization before error diagnosis. Such minimization techniques can remove redundant information from the bug trace and greatly facilitate error diagnosis. We adopted one such technique [8] to minimize the traces for DLX and Alpha, and the length of the original traces is shown in parentheses. In general, one trace is enough to localize the errors to a small number of symptom cores, while additional traces may further reduce this number.

Comparison between RTL and gate-level error diagnosis: for comparison with previous work, we also synthesized part of the larger benchmarks and performed gate-level error diagnosis using Ali's [3] sequential error-diagnosis techniques described in Section 2.2, and the results are summarized in Table 4. Due to circuit unrolling, the generated CNFs may contain more than 10 million clauses, producing very difficult error-diagnosis instances. This comparison clearly indicates that diagnosing functional errors at the RTL has significant advantages over the gate level: shorter runtime and more accurate diagnoses. As Table 3 shows, most errors

Bench-mark	Bug ID	Errors found		Runtime (sec)
		#Sites	#Cores	
MRISC	A	**Time-out (48 hours)**		
DLX	A	**Out of memory**		
	B	**Out of memory**		
	C	1	170	**34,829**
	D	1	6	**49,787**
	E	1	193	**19,621**
	F	**Out of memory**		
Alpha	A	**Time-out (48 hours)**		
	B	**Time-out (48 hours)**		
	C	**Out of memory**		

Table 4: Gate-level error-diagnosis results. "#Sites" is the number of *error sites* reported in each core, and "#Cores" is the total number of distinct symptom cores.

can be diagnosed using our techniques within a few minutes, while Table 4 shows that identifying the same errors at the gate level takes more than 48 hours in many cases. One major reason for this is that the number of possible symptom variables (error sites), i.e., internal netlist signals responsible for the bug, is significantly smaller in RTL diagnosis. This is due to the fact that one simple RTL statement may be synthesized into a complex netlist, which proliferates the number of error sites. For example, a statement like "a = b + c" creates only one symptom variable at the RTL. Its synthesized netlist, however, may contain hundreds of error sites, depending on the implementation of the adder and the bit-width of the signals. The small number of potential symptom variables at the RTL significantly reduces the search space for PB or SAT solvers and provides very short diagnosis runtime, even though the instance sizes generated by the gate-level and RTL diagnosis are similar. These results clearly indicate that adopting gate-level techniques into RTL is a winning approach: it provides excellent accuracy because formal analysis can be performed, yet it does not have any drawback common in gate-level analysis in that it is still highly scalable and efficient. This is achieved by our new constructs that model errors at the RTL instead of the gate level. These results also demonstrate that trying to diagnose RTL errors at the gate level and mapping the results back to RTL is ineffective and inefficient, not to mention the fact that such a mapping is usually difficult to find.

Case study: we use DLX(D) as an example to show the power of our error-diagnosis techniques. Its RTL code is shown below:

```
always@(memstage or exstage or idstage or rs3rd or rs3rt or
rs4rd or rs4rt or rsr31)
  casex ({memstage,exstage,idstage,rs3rd,rs3rt,rs4rd,rs4rt,rsr31})
    {‘ALUimm, ‘dc3, ‘dc3,‘dc,‘dc, ‘dc, ‘true,‘dc}:
    RSsel = ‘select_stage3_bypass; // Buggy
```

In this example, the buggy code selects stage3 bypass, while the correct implementation should select stage4. Error diagnosis returns two symptom cores: *RSsel* and *ALUout*. Obviously, *RSsel* is the correct diagnosis. However, *ALUout* is also a correct diagnosis because if the ALU can generate correct outputs even though the control signal is incorrect, then the bug can also be fixed. However, this is not a desirable fix. This case study shows that REDIR can suggest various ways to repair the same error, allowing the designer to consider different possibilities in order to choose the best one.

5.2 Simulation-based Error Diagnosis

In this experiment, we performed simulation-based diagnosis using the algorithm described in Section 3.3 with Insight, an experimental RTL symbolic simulator from [24]. Benchmarks Pipe and CF_FFT were used in this experiment. Simulation took 23.8 and 162.9 seconds to generate SAT instances for these benchmarks, respectively. The SAT solver included in Insight then solved the instances in 1 and 723 seconds respectively, and it successfully identified the design errors. Note that currently, the SAT solver only returns one, instead of all possible symptom cores. Although the runtime of simulation-based approach is longer than the synthesis-based method, it does not require the design to be synthesized in advance, thus saving the synthesizer runtime.

5.3 Error Correction

In our error-correction experiment, we applied the techniques described in Section 4 to fix those same errors diagnosed in Table 3. We used combinational diagnosis in this experiment, and derived a corrected solution using the synthesis approach described in [7]. We summarize the results in Table 5 where we report which of the two synthesis techniques in [7] we used, either GDS or EGS. GDS involves an exhaustive search of the solution space, and finds solutions with minimal number of logic operations; while EGS is approximate but faster. Note that the resynthesis tool can be replaced easily by other tools, such as Espresso or MVSIS. In the table, "#Cores fixed" is the number of symptom cores that can be corrected using our error-correction techniques, and "#Fixes" is the number of distinct repair solutions. We applied GDS first in the experiment, and observed that GDS often returns a large number of valid fixes that can correct the design errors. One reason is that GDS performs exhaustive search to find new logic expressions; therefore, it may find many different ways to produce the same signal. For example, "$A \cdot \overline{B}$" and "$A \cdot (A \oplus B)$" are both returned even though they are equivalent. Another reason is that we only diagnosed short bug traces, which may produce spurious fixes: signatures of different variables are the same even though their functions are different. To overcome this issue, we sort the fixes and consider first those entailing fewer logic operations. For some benchmarks (DLX C-F and Alpha), GDS exhausts all the memory available due to the exhaustive nature of its search approach. In those situations we resorted to EGS, which returns only one possible correction solution per symptom core as reflected in the table.

Benchmark	Bug ID	#Cores fixed	Resyn. method	#Fixes	Runtime (sec)
Pipe	A	1	GDS	2214	1.0
Pre_norm	A	1	GDS	4091	1.1
	B	1	GDS	4947	2.4
	C	1	GDS	68416	5.6
	D	2	GDS	79358	7.1
	E	3	GDS	548037	41.6
MD5	A	1	GDS	33625	4.1
	B	0	GDS	0	3.86
CF_FFT	A	3	GDS	214800	141.6
DLX	A	0	GDS	0	1.3
	B	3	GDS	5319430	111.2
	C	5	EGS	5	1.6
	D	3	EGS	3	1.6
	E	4	EGS	4	1.4
	F	2	EGS	2	2.9
	G	1	GDS	51330	0.7
Alpha	A	5	EGS	5	7.9
	B	4	EGS	4	10.4
	C	3	EGS	3	8.5

Table 5: Error-correction results. Combinational diagnosis is used in this experiment.

Table 5 shows that we could not find valid fixes for benchmarks MD5(B) and DLX(A). The reason is that the bugs in these benchmarks involve multi-bit variables. For example, bug MD5(b) is an incorrect state transition for a 3-bit state register. Since in this experiment we only consider the least-significant bits of such variables during error correction, we could not find a valid fix. This problem can be solved by inserting a conditional assignment for every bit in a multi-bit variable.

5.4 Discussion of Experimental Results

The error-diagnosis results show that our error-modeling construct and diagnosis techniques can effectively localize design errors to a small number of symptom variables. On the other hand, our error-correction results suggest that options to repair the diagnosed errors abound. The reason is that the search space of error correction is much larger than error diagnosis: there are several ways to synthesize a logic function. As a result, finding high-quality fixes for a bug requires much more information than providing high-quality diagnoses. Although this can be achieved by diagnosing longer or more numerous bug traces, the runtime of REDIR will also increase.

This observation shows that automatic error correction is a much more difficult problem than automatic error diagnosis. In practice, however, engineers often find error diagnosis more difficult than error correction. It is common that engineers need to spend days or weeks finding the cause of a bug. However, once the bug is identified, fixing it may only take a few hours. To this end, our error-correction technique can also be used to facilitate manual error repair, and it works as follows: (1) the engineer fixes the RTL code manually to provide new logic functions for the symptom cores identified by error diagnosis; and (2) REDIR simulates the new functions to check whether the signatures of symptom cores can be generated correctly using the new functions. If the signatures cannot be generated by the new functions, then the fix is invalid. In this way, engineers can check the correctness of their fixes before running verification, which can accelerate the manual error-repair process significantly.

The synthesis-based results show that our techniques can effectively handle designs as large as 2000 lines of RTL code, which is approximately the size that an engineer actively works on. Since synthesis tools are available in most companies, REDIR can be used by engineers everyday to facilitate their debugging process. For example, REDIR can be used at early design stages where design representations are still small and correct output responses can be determined manually, as well as at later design stages where circuits are larger and correct output responses are generated by high-level models. On the other hand, the simulation-based results suggest that our techniques are promising. Once RTL symbolic simulators become accessible to most companies, REDIR can exploit their simulation power to handle even larger designs.

In our experiments we present results using designs as large as 127K cells with traces that can be hundred-cycles long, which are more than 10X larger than the benchmarks reported in other RTL error-diagnosis literature [11, 16, 18, 19, 20]. In addition, we evaluated REDIR using designs with real bugs and successfully identified the RTL signals responsible for the bugs. These results show that our techniques can be applied to more complex designs than existing solutions can tackle.

6. CONCLUSIONS

In this paper we proposed several constructs and algorithms that provide a new way to diagnose and correct errors at the RTL, including: (1) an RTL error modeling construct; (2) scalable error-diagnosis algorithms using Pseudo-Boolean constraints, synthesis, and simulation; and (3) a novel error-correction technique using signatures. To empirically validate our proposed techniques, we developed a novel verification framework, called REDIR. To this end, our experiments with industrial designs demonstrate that REDIR is efficient and scalable. In particular, designs up to a few thousand lines of code (or 100K cells after synthesis) can be diagnosed within minutes with high accuracy. Since our methods only rely on correct output responses and provide support for both combinational and sequential circuits, they can be deployed in most verification methodologies. We believe that because of its qualities and its ease of deployment REDIR could fundamentally change the practice of RTL debugging.

Acknowledgments. We thank Chilai Huang and Zhihong Zeng (Avery Design Systems) for their support on the Insight product.

7. REFERENCES

[1] M. S. Abadir, J. Ferguson and T. E. Kirkland, “Logic Verification via Test Generation”, *IEEE TCAD*, pp. 138-148, Jan. 1988.

[2] M. F. Ali, S. Safarpour, A. Veneris, M. Abadir and R. Drechsler, “Post-Verification Debugging of Hierarchical Designs”, *ICCAD*, 2005, pp. 871-876.

[3] M. F. Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith and M. Abadir, “Debugging Sequential Circuits Using Boolean Satisfiability”, *ICCAD*, 2004, pp. 44-49.

[4] V. Bertacco, “Scalable Hardware Verification with Symbolic Simulation”, Springer, 2005.

[5] R. E. Bryant, D. Beatty, K. Brace, K. Cho and T. Sheffler, “COSMOS: a compiled simulator for MOS circuits”, *DAC*, 1987, pp. 9-16.

[6] R. Bloem and F. Wotawa, “Verification and Fault Localization for VHDL Programs”, *Journal of the Telematics Engineering Society (TIV)*, pp. 30-33, Vol. 2, 2002.

[7] K.-H. Chang, I. L. Markov and V. Bertacco, “Fixing Design Errors with Counterexamples and Resynthesis”, *ASPDAC*, 2007, pp. 944-949.

[8] K.-H. Chang, V. Bertacco and I. L. Markov, “Simulation-based Bug Trace Minimization with BMC-based Refinement”, *ICCAD*, 2005, pp. 1045-1051.

[9] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *Proc. Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.

[10] N. Eén and N. Sörensson, “Translating Pseudo-Boolean Constraints into SAT,” in *JSAT*, 2006, pp. 1-25.

[11] T.-Y. Jiang, C.-N. J. Liu and J.-Y. Jou, “Estimating Likelihood of Correctness for Error Candidates to Assist Debugging Faulty HDL Designs,” *ISCAS*, 2005, pp. 5682-5685.

[12] A. Kolbl, J. Kukula and R. Damiano, “Symbolic RTL Simulation,” *DAC’01*, pp. 47-51.

[13] A. Kolbl, J. Kukula, K. Antreich and R. Damiano, “Handling Special Constructs in Symbolic Simulation”, *DAC’02*, pp. 105-110.

[14] A. Kuehlmann, D. I. Cheng, A. Srinivasan and D. P. Lapotin, “Error Diagnosis for Transistor-Level Verification”, *DAC’94*, pp. 218-224.

[15] J. C. Madre, O. Coudert and J. Pl. Billon, “Automating the Diagnosis and the Rectification of Design Errors with PRIAM”, *ICCAD’89*, pp. 30-33.

[16] J.-C. Rau, Y.-Y. Chang and C.-H. Lin, “An Efficient Mechanism for Debugging RTL Description”, *IWSOC*, 2003, pp. 370-373.

[17] R. Rudell and A. Sangiovanni-Vincentelli, “Multiple-valued minimization for PLA optimization”, *IEEE TCAD*, pp. 727-750, Sep. 1987.

[18] C.-H. Shi and J.-Y. Jou, “An Efficient Approach for Error Diagnosis in HDL Design”, in *Proc. ISCAS*, 2003, pp. 732-735.

[19] S. Staber, B. Jobstmann and R. Bloem, “Finding and Fixing Faults”, *Springer-Verlag LNCS 3725*, 2005, pp. 35-49.

[20] S. Staber, G. Fey, R. Bloem and R. Drechsler, “Automatic Fault Localization for Property Checking”, *Springer-Verlag LNCS 4383*, 2007, pp. 50-64.

[21] A. Smith, A. Veneris and A. Viglas, “Design Diagnosis Using Boolean Satisfiability”, *ASPDAC*, 2004, pp. 218-223.

[22] A. Veneris and I. N. Hajj, “Design Error Diagnosis and Correction via Test Vector Simulation”, *IEEE TCAD*, Dec. 1999, pp. 1803-1816.

[23] Y.-S. Yang, S. Sinha, A. Veneris and R. Brayton, “Automating Logic Rectification by Approximate SPFDs”, *ASPDAC*, 2007, pp. 402-407.

[24] http://www.avery-design.com/

[25] “Bug UnderGround”, http://bug.eecs.umich.edu/

[26] http://www.opencores.org/

[27] http://www.si2.org/