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 the presence of 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 designs up to several thousand lines of RTL code in minutes with significantly higher accuracy than previous gate-level centered solutions. 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 the ability of developers 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 [6, 14, 19, 20, 21] or the transistor level [13]. However, most debugging effort occurs in the Register-Transfer Level (RTL) description of a circuit, where design activities are carried out. The lack of powerful and automatic tools for error diagnosis and correction at this level greatly reduces designers' productivity when fixing even very simple design 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 [10, 15, 17] employ 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 and cannot automatically correct the errors. The second group of techniques, such as [5], use formal analysis of an HDL description and failed properties; because of that these techniques can only be deployed in a formal verification framework, and cannot be applied in a simulation-based verification flow common in the industry today. In addition, these techniques cannot repair identified errors automatically. Finally, the work by Staber *et al.* [18] 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. In addition, 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 available is a high-level model, often written in a high-level language, which produces the correct I/O behavior of the system.

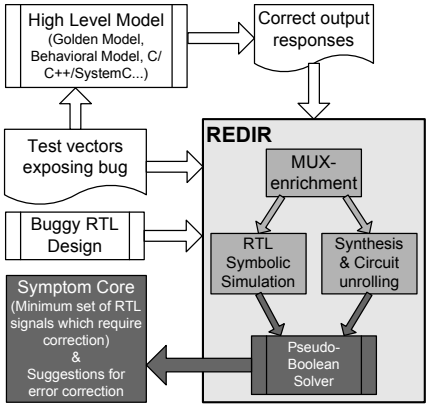

Figure 1: REDIR framework. Inputs to the tool are an RTL design (which includes one or more errors), test vectors exposing the bug(s), and correct output responses for those vectors obtained from a high-level simulation. Outputs of the tool include REDIR *symptom core* (a minimum cardinality set of RTL 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 take a completely different approach by adopting 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. Moreover, 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 for error diagnosis, as opposed to previous solutions that use MUXes implicitly; (2) innovative error-diagnosis algorithms using synthesis or symbolic simulation; and (3) a novel error-correction technique using signal behaviors (*signatures*) 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 techniques in a framework called *REDIR* (RTL Error DIagnosis and Repair), highlighted in Figure 1. The inputs to the framework include a design containing one or more bugs, a set of test 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 output of the framework is 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 fixes of the signals in the *symptom core* to help a designer correct those signals. 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 size that an engineer actively works on. As a result, REDIR can assist engineers in their everyday debugging tasks and fundamentally accelerate the RTL design process.

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 Pseudo-Boolean problems, which are an extension of SATisifiability 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. We then show the basic idea behind symbolic simulation, which we use as an alternative, compact way to formulate the PB problem. Finally, we briefly describe signature-based resynthesis techniques that will be used in our error correction.

2.1 Pseudo-Boolean Problems

PB problems, also called 0-1 integer linear programming problems, are an extension of SATisfiability 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+ [9]).

2.2 Gate-level Error Diagnosis Techniques

Gate-level error diagnosis and correction techniques have been studied extensively in the past. Early work often relies on specific error models to simplify the problem, such as [1, 14]. Recently, the power and effectiveness of gate-level error diagnosis have been improved by the work of Smith *et al.* [19], which does not rely on any error model. In Smith's error-diagnosis 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.* [2] extended Smith's work by unrolling the circuit, before the CNF conversion step, M times, where M is the sequential length of the given trace. Similar approach is used in our synthesis-based diagnosis. As we show in our experimental results, however, 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 [3, 4]. 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 [11, 12]. Symbolic simulation is an alternative way to generate an instance of the Pseudo-Boolean constraint problem that we use in our error diagnosis framework.

2.4 Signature-based Resynthesis Techniques

Our error correction technique is based on signatures, where a signature is essentially a signal's partial truth table and represents a signal's behavior. The error-correction problem in REDIR is formulated as follows: given a target signature and a collection of candidate signatures, find a logic expression that generates the target signature using the candidate signatures. Since signatures are partial truth tables, this process is essentially a logic synthesis step, and any synthesis tool could be used for this purpose. Our previous work that is specially tuned for resynthesizing from partial truth tables was published in [6]. The work proposes two techniques: one involves an exhaustive search of the solution space, and finds solutions with minimal number of logic operations (GDS). The other method is approximate but faster (EGS).

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 (wire, 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, usually generated by a high-level behavioral model. The objective of the error diagnosis is to identify a minimal number of variables in the RTL description that are responsible for the design's erroneous behavior. Moreover, by modifying the logic of those variables, the design errors can be corrected. Each signal found to affect the correctness of the design is called a *symptom variable*. Without minimization, the set of *symptom variables* reported would include the root cause of the bug and 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 solution as close to the root cause of the erroneous behavior as possible. On the other hand, if minimizing the number of *symptom variables* is not preferred (e.g. one-output circuits), a solution with a specified number N of *symptom variables* can be found by converting the PBC to a SAT instance with the total number of *symptom variables* set to N.

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 these conditional assignments are used for error diagnosis only and should not appear in the final synthesized design. However, they 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 | b". Obviously, to produce the correct output that we obtain from a high-level model, 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 | 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 pseudocode 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)
1. create a new signal wire v_n and new inputs v_f and v_{sel};
2. add conditional assignment "$v_n = v_{sel}$? v_f : v";
3. 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.

3.2 Diagnosis with Synthesis

After the error-modeling constructs have been inserted into a design, error diagnosis is used to identify the minimal number of *select variables* that should be asserted along with the values of their corresponding *free variables* to produce the correct circuit behavior. 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. In this way, the synthesized netlist faithfully preserves the constructs inserted at the RTL, enabling accurate RTL error diagnosis. This is significantly different from diagnosing design 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*);
1 *CNF* = unroll *designCNF* c times;
2 connect all *select variables* in CNF to those in the first cycle;
3 constrain PI/PO in CNF using *inputs*/*outputs*;
4 *PBC* = *CNF*, min($\sum$ *select variables*);
5 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 variables* 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 scales further than the synthesis-based technique. We achieve this 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 Pseudo-Boolean 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 [11, 12]. 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 [22].

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  ∀i, 1 ≤ i ≤ n, v_{i_sel} = new_symbol();
2  for t = 1 to c begin // Simulate c cycles
3     PI = test_vector at cycle t;
4     ∀i, 1 ≤ i ≤ n, v_{i_f@t} = new_symbol();
5     PO_{@t} = simulate(design);
6  end
7  PBC = ⋀_{j=1}^{o} ⋀_{t=1}^{c} (PO_{j@t} = CPO_{j@t}), min(Σ_{i=1}^{n} 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.

Below we present an example of a buggy design to illustrate the symbolic simulation-based error-diagnosis technique.

Example 1. Assume that the circuit shown in Figure 6 contains an error: signal g_1 is erroneously assigned to expression "r1 | r2" instead of "r1 & r2". Conditional assignments, highlighted in boldface, have been inserted into the circuit using the techniques described in Section 3.1. For simplicity reasons, we do not include the MUXes at the outputs of registers *r1* and *r2*. The trace that exposes the error in two simulation cycles consists of the following values for inputs {I1, I2}: {0, 1}, {1, 1}. When the same trace is simulated by a high-level behavioral model, the correct output responses for {O1, O2} are generated: {0, 0}, {1, 0}. Besides these output responses, no addition information, such as values of internal signals and registers, is required. We annotate the symbols injected during the simulation by their cycle numbers using subscripts. The Boolean expressions for the primary outputs for the two cycles of simulation are:

$O1_{n@1} = O1_{sel}\ ?\ O1_{f@1} : [I1_{@1} \mid (g1_{sel}\ ?\ g1_{f@1} : 0)]$
$O2_{n@1} = O2_{sel}\ ?\ O2_{f@1} : [I2_{@1}\ \&\ (g1_{sel}\ ?\ g1_{f@1} : 0)]$
$O1_{n@2} = O1_{sel}\ ?\ O1_{f@2} : \{I1_{@2} \mid [g1_{sel}\ ?\ g1_{f@2} : (I1_{@1}\ \&\ I2_{@1})]\}$
$O2_{n@2} = O2_{sel}\ ?\ O2_{f@2} : \{I2_{@2}\ \&\ [g1_{sel}\ ?\ g1_{f@2} : (I1_{@1}\ \&\ I2_{@1})]\}$

Since the primary inputs are scalar values, the expressions can be greatly simplified during symbolic simulation. For example, we know that $I1_{@2} = 1$; therefore, $O1_{n@2}$ can be simplified to $O1_{sel}\ ?\ O1_{f@2} : 1$. As a result, the Boolean expressions actually generated by the symbolic simulator are:

$O1_{n@1} = O1_{sel}\ ?\ O1_{f@1} : (g1_{sel}\ ?\ g1_{f@1} : 0)$
$O2_{n@1} = O2_{sel}\ ?\ O2_{f@1} : (g1_{sel}\ ?\ g1_{f@1} : 0)$
$O1_{n@2} = O1_{sel}\ ?\ O1_{f@2} : 1$
$O2_{n@2} = O2_{sel}\ ?\ O2_{f@2} : (g1_{sel}\ ?\ g1_{f@2} : 0)$

To perform error diagnosis, we constrain the output expressions using the correct responses, and then construct a PBC as follows:

$PBC = (O1_{n@1} = 0) \wedge (O2_{n@1} = 0) \wedge (O1_{n@2} = 1) \wedge (O2_{n@2} = 0)$,
$\min(O1_{sel} + O2_{sel} + g1_{sel})$.

One possible solution of this PBC is to assert $g1_{sel}$, which provides a correct *symptom core*.

```
module example(clk, I1, I2, O1_n, O2_n, g1_sel, O1_sel,
O2_sel, g1_f, O1_f, O2_f);
  input I1, I2, g1_sel, O1_sel, O2_sel, g1_f, O1_f, O2_f
  output O1_n, O2_n;
  reg r1, r2;
  initial begin r1= 0; r2= 0; end
  always @(posedge clk) begin
    r1= I1; r2= I2;
  end
  assign g1 = r1 | r2;
  assign O1 = I1 | g1_n;
  assign O2 = I2 & g1_n;
  assign g1_n= g1_sel ?  g1_f :  g1;
  assign O1_n= O1_sel ?  O1_f :  O1;
  assign O2_n= O2_sel ?  O2_f :  O2;
endmodule
```

Figure 6: Design for the example. Wire g1 should be driven by "r1 & r2", but it is erroneously driven by "r1 | r2". The changes made during MUX-enrichment are marked in boldface.

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, the *select variables* of instances of the same module should be shared, while the *free variables* should not. This is because all instances of the same module will 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, it is possible that a bug requires fixing only one RTL instance while other instances of the same module can be left intact. This situation requires generation of new RTL modules and is currently not handled by our diagnosis techniques.

4. RTL ERROR CORRECTION

The RTL error-correction problem is formulated as follows: given an erroneous RTL description of a digital design, find a variant description for one or more of the modules that compose it so that the new design presents a correct behavior for the errors, while leaving the known-correct behavior unchanged. Although many error-repair techniques exist for gate-level designs, very few studies focus on the RTL. One major reason is the lack of logic representations that can support the logic manipulation required during RTL error correction. For example, the logic of a signal in a gate-level netlist can be easily represented by BDDs, and modifying the function of the signal can be supported by the manipulation of its BDDs. However, most existing logic representations cannot be easily applied to an RTL variable. This problem is further exacerbated by the fact that an RTL module may be instantiated multiple times, creating many different functions for an RTL variable depending on where it is instantiated.

In [6] we proposed a framework for gate-level error correction. Our approach utilizes only signatures, which can be easily calculated via simulation, making our techniques especially suitable for RTL error correction. However, techniques in [6] could be applied only to combinational circuits, and could handle design hierarchies. To support the error-correction requirements at the RTL, where most designs contain hierarchies and are sequential, we propose a new error-correction scheme based on similar concepts. In this section, we first describe the baseline error-correction technique that is easier to understand. Next, we show how signatures should be generated at the RTL to handle hierarchical and sequential designs. Finally, we provide some insights that we obtained during the implementation of our system.

4.1 Baseline Error Correction Technique

For a flattened combinational design, error correction is performed as follows: (1) signatures of RTL variables are generated using simulation; (2) error diagnosis is performed to find a *symptom core*; (3) signatures of the *symptom variables* in the *symptom core* are replaced by the values of their corresponding *free variables*; and (4) synthesis is applied to find logic expressions generating the signatures of the *symptom variables*. By replacing the expressions that generate the functions of the *symptom variables* with those new expressions, design errors can be corrected.

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 in order to correct the design's behavior. Since changing an RTL module will affect all the instances produced by the module, we concatenate the simulation values of the *instance variables* derived from the same *module variable* to produce the signature for the *module variable*. This way, we can guarantee that a change in a module will affect instances 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 7. 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;
  child c1(), c2(), c3();
endmodule
module child;
  wire v;
endmodule
```

Simulation values:

Cycle 0: top.c1.v = 0, top.c2.v = 0, top.c3.v = 1

Cycle 1: top.c1.v = 1, top.c2.v = 0, top.c3.v = 0

Constructed signature for RTL error correction:

child.v = $\overbrace{\underbrace{1}_{c1.v}\ \underbrace{0}_{c2.v}\ \underbrace{0}_{c3.v}}^{cycle\ 1}\ \overbrace{\underbrace{0}_{c1.v}\ \underbrace{0}_{c2.v}\ \underbrace{1}_{c3.v}}^{cycle\ 0}$

Figure 7: Signature-construction example. Simulation values of variables created from the same RTL variable at all cycles should be concatenated for error correction.

Example 2. Using the same circuit as Example 1. The values returned by the PB solver for $g1_{f@0}$ and $g1_{f@1}$ are both 0. Since the inputs to g_1 are $\{0, 0\}$ and $\{0, 1\}$ for the first two cycles, the correct expression for g_1 should generate 0 for these two inputs. RTL error correction returns the following new logic expressions that can fix the error: $g1 = r1\&r2$, $g1 = r1$, etc. Note that although the correct fix is returned, the fix is not unique. In general, longer traces containing various test vectors will identify the error with higher precision and suggest better fixes than short ones.

4.3 Implementation Insights

Fixing errors involving multi-bit variables is more difficult than fixing errors involving only one-bit variables because different bits in the variable may be generated differently. To solve this problem, we allow the user to insert a conditional assignment for each bit in the variable. Alternatively, REDIR can also be configured to consider only the least-significant bit when performing error correction. This is useful when the variable is considered as a whole.

In synthesis-based error diagnosis, we observe that it is difficult to identify the wires derived from the same RTL variable in a synthesized netlist. To overcome this problem, we add the outputs of inserted conditional statements to the primary outputs of the MUX-enriched modules to obtain the simulated values of the RTL variables. To improve our error-correction quality, we utilize observability don't-cares in our synthesis-based approach by simulating the complement signatures of *symptom variables* and observe the changes at primary outputs (including inputs to registers).

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 [24] with RTL Compiler v4.10 from Cadence as the synthesis tool. For simulation-based diagnosis, we adopted an experimental RTL symbolic simulator, Insight 1.4, from Avery Design Systems [22]. For efficiency, we implemented the techniques described in [9] to convert PB problems to SAT problems and adopted MiniSAT as our SAT solver [8]. All the experiments were conducted on an AMD Opteron 880 (2.4GHz) Linux workstation with 16GB memory. The designs under test included several circuits selected from OpenCores [23] (Pre_norm, MD5, MiniRISC, and CF_FFT), the picoJava-II microprocessor (Pipe), DLX, and Alpha. Bugs (described in Table 2) were injected into these benchmarks, with the exception of

Benchmark	Description	#Flip-flops	Trace type	Gate-level [2, 19]		RTL (Ours)	
				#Cells	#MUXes	#Lines	#Assign
Pipe	Part of PicoJava pipeline control unit	2	Constrained-random	55	72	264	31
Pre_norm	Part of FPU	71	Constrained-random	1877	1877	270	43
MD5	MD5 full chip	910	Direct test	13311	13313	438	37
MiniRISC	MiniRISC full chip	887	Direct test	6402	6402	2013	43
CF_FFT	Part of the CF_FFT chip	16,638	Constrained-random	126532	126560	998	223
DLX	5-stage pipeline CPU running MIPS-Lite ISA	2,062	Constrained-random	14725	14727	1225	84
Alpha	5-stage pipeline CPU running Alpha ISA	2,917	Constrained-random	38299	38601	1841	134

Table 1: Characteristics of benchmarks. "#MUXes" is the number of MUXes inserted by gate-level diagnosis [2, 19] for comparison, and "#Assign" is the number of conditional assignments inserted by our solution.

DLX and Alpha, which already included bugs. We used constrained-random simulation to generate bug traces for Pipe, Pre_norm, and CF_FFT, while the bug traces for the rest of the benchmarks were generated using the verification environment shipped with the designs. Traces to expose bugs in DLX and Alpha were given by the verification engineer and were generated using a constrained-random simulation tool. The characteristics of these benchmarks are summarized in Table 1. In the table, "RTL #Lines" is the number of lines of RTL code in a design, and "Gate-level #Cells" is the cell count of the synthesized netlist. To compare our results with previous work, we implemented the algorithms for gate-level error diagnosis in [2, 19]. In the table, we list the number of MUXes inserted by their techniques in column "#MUXes", and the number of conditional assignments under "#Assign".

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 bugs in benchmarks. DLX and Alpha include native bugs, while the others were manually injected.

5.1 Synthesis-based Error Diagnosis

In this experiment, we performed combinational and sequential error diagnosis using the synthesis-based techniques described in Section 3.2. For comparison with previous work, we also synthesized the benchmarks and performed gate-level error diagnosis using Smith's and Ali's [2, 19] techniques described in Section 2.2. The results are summarized in Table 3. Recall that a *symptom core* suggests a possible set of signals to modify for correcting 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 benchmarks. In other words, REDIR accurately pointed out the signals that exhibited incorrect behavior.

Comparison between RTL and gate-level error diagnosis: 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 can be diagnosed using our techniques within a few minutes, while 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, as can be observed from the numbers of inserted conditional assignments shown in Table 1. 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. In addition, one bug at the RTL may transform into multiple simultaneous bugs at the gate level. Since runtime of error diagnosis grows substantially with each additional bug [19], being able to diagnose errors at the RTL avoids the expensive multi-error diagnosis process at the gate level. We also observed that although the runtime of the RTL error diagnosis still increases with each additional bug, its growth rate is much smaller than the growth rate at the gate level. For example, as Table 3 shows, the runtime of the gate-level diagnosis for Pre_norm(A) and (D), which combined (A) and (B), was 63.6 and 88.7 seconds, respectively. For RTL diagnosis, the runtime was 13.2 and 13.8 seconds, respectively. These results clearly indicate that adopting gate-level techniques into RTL is the correct 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.

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 cy-

Bench-mark	Bug ID	Bug traces		Gate-level diagnosis [2, 19]						RTL diagnosis (Our work)					
		#tra-ces	#cy-cles	Combinational			Sequential			Combinational			Sequential		
				Errors found		Runtime	Errors found		Runtime	Errors found		Runtime	Errors found		Runtime
				#Sites	#Cores	(sec)	#Sites	#Cores	(sec)	#Symp.	#Cores	(sec)	#Symp.	#Cores.	(sec)
Pipe	A	32	200	1	1	6.9	1	1	**7.1**	1	1	6.0	1	1	**6.0**
Pre_ norm	A	32	20	1	1	51.1	1	1	**63.6**	1	1	13.2	1	1	**13.2**
	B	32	20	1	3	41.6	1	4	**46.7**	1	1	11.4	1	2	**13.4**
	C	32	20	**Time-out (48 hours) with > 10 error sites**						1	1	11.4	1	1	**11.4**
	D	32	20	2	3	73.3	2	4	**88.7**	2	1	12.4	2	2	**13.8**
	E	32	20	**Time-out (48 hours) with > 8 error sites**						3	2	13.9	3	4	**17.4**
MD5	A	1	200	**Time-out (48 hours) with > 6 error sites**						1	1	83.3	1	3	**173.2**
	B	1	200	1	2	10,980	1	4	**41,043**	1	1	42.9	1	2	**110.1**
	C	1	50	1	3	2,731	1	28	**17,974**	1	1	14.1	1	6	**49.8**
MRISC	A	1	500	States unavailable			**Time-out (48 hours)**			States unavailable			1	2	**32.0**
CF_FFT	A	32	15	1	1	109,305	Trace unavailable			1	4	364.8	Trace unavailable		
DLX	A	1	150	**Time-out (48 hours)**			**Out of memory**			1	1	41.2	1	3	**220.8**
	B	1	68 (178)	1	20	15,261	**Out of memory**			1	4	54.8	1	17	**1886.3**
	C	1	47 (142)	1	45	11,436	1	170	**34,829**	1	5	15.8	1	11	**104.0**
	D	1	77 (798)	1	6	18,376	1	6	**49,787**	1	3	27.5	1	9	**2765.1**
	E	1	49 (143)	1	12	9743.5	1	193	**19,621**	1	4	19.1	1	12	**105.2**
	F	1	188	1	10	15,184	**Out of memory**			1	2	67.8	1	2	**457.4**
	G	1	30 (1080)	1	9	4160.1	Trace unavailable			1	1	11.3	Trace unavailable		
Alpha	A	1	70(256)	**Time-out (48 hours)**						1	5	127.4	1	9	**525.3**
	B	1	83(1433)	**Time-out (48 hours)**						1	5	111.6	1	5	**368.9**
	C	1	150(9950)	**Out of memory**						1	3	122.3	1	3	**250.5**

Table 3: Synthesis-based error diagnosis results. For gate level, "#Sites" is the number of *error sites* reported in each core, and for RTL "#Symp." is the number of *symptom variables* in each core. '#Cores" is the total number of *symptom cores* returned by either approach. The results show that RTL diagnosis outperforms gate-level diagnosis in all the benchmarks: the runtime is shorter, and the diagnosis is more accurate. Bug traces for several DLX/Alpha benchmarks have been minimized before diagnosis, and their original lengths are shown in parentheses.

cles 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 [7] 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.

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 [22]. 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.

Bench-mark mark	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 4: Error correction results. Combinational diagnosis is used in this experiment.

5.3 Error Correction

In our error-correction experiment, we applied the techniques described in Section 4 to fix the errors diagnosed in Table 3. We used combinational diagnosis in this experiment, and corrected the error location using the synthesis tool in [6]. We summarized the results in Table 4 where we indicate which of the two synthesis techniques in [6] we used, either GDS or EGS (see Section 2.4). 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 ways to fix the errors. 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. As a result, we only report the first 100 fixes in our implementation, where the fixes are sorted so that those with smaller number of logic operations are returned first. Due to the exhaustive-search nature of GDS, memory usage of GDS may be high during the search, as are the cases for benchmarks DLX (C-F) and Alpha. In these benchmarks, GDS ran out of memory, and we relied on EGS to find fixes that can correct the errors. Since EGS only returns one logic expression when fixing an error, the number of possible fixes is significantly smaller.

Table 4 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 various 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. 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 automatically exploit their simulation power to handle even larger designs.

6. CONCLUSIONS

In this paper we proposed several constructs and algorithms that provide a fundamentally 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 support both combinational and sequential circuits, they can be applied to various designs in all mainstream verification flows. The superior scalability, efficiency and accuracy ensure that REDIR can be used by engineers in their everyday debugging tasks, which can fundamentally change the RTL debugging process.

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, A. Veneris, S. Safarpour, R. Drechsler, A. Smith and M. Abadir, "Debugging Sequential Circuits Using Boolean Satisfiability", *ICCAD*, 2004, pp. 44-49.

[3] V. Bertacco, "Scalable Hardware Verification with Symbolic Simulation", Springer, 2005.

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

[5] 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.

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

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

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

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

[10] 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.

[11] A. Kolbl, J. Kukula and R. Damiano, "Symbolic RTL Simulation," *DAC'01*, pp. 47-51.

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

[13] A. Kuehlmann, D. I. Cheng, A. Srinivasan and D. P. Lapotin, "Error Diagnosis for Transistor-Level Verification", *DAC'94*, pp. 218-224.

[14] 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.

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

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

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

[18] S. Staber, B. Jobstmann and R. Bloem, "Finding and Fixing Faults", in *CHARME*, Springer-Verlag LNCS 3725, 2005, pp. 35-49.

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

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

[21] Y.-S. Yang, S. Sinha, A. Veneris anr R. Brayton, "Automating Logic Rectification by Approximate SPFDs", *ASPDAC*, 2007, pp. 402-407.

[22] `http://www.avery-design.com/`

[23] `http://www.opencores.org/`

[24] `http://www.si2.org/`