Fixing Design Errors with Counterexamples and Resynthesis

Kai-Hui Chang, Igor L. Markov and Valeria Bertacco
University of Michigan, Ann Arbor, MI 48109
{changkh, imarkov, valeria}@umich.edu

Abstract — In this work we propose a new error-correction framework, called CoRé, which uses counterexamples, or bug traces, generated in verification to automatically correct errors in digital designs. CoRé is powered by two innovative resynthesis techniques, Goal-Directed Search (GDS) and Entropy-Guided Search (EGS), which modify the functionality of internal circuit's nodes to match the desired specification. We evaluate our solution to designs and errors arising during combinational equivalence-checking, as well as simulation-based verification of digital systems. Compared with previously proposed techniques, CoRé is more powerful in that: (1) it can fix a broader range of error types because it does not rely on specific error models; (2) it derives the correct functionality from simulation vectors, hence not requiring golden netlists; and (3) it can be applied to a range of verification flows, including formal and simulation-based.

I. Introduction

The vast and growing complexity of silicon designs being developed today poses a number of challenges on verifying that such designs fully deliver the expected functionality. Much effort has been dedicated in the past decade to improve the quality, performance, scalability and automation of the functional verification process. However, the goal of manufacturing correct designs is still far from achievable: on the one hand industry teams often dedicate more than two thirds of their resources to verification, but on the other hand, studies report that functional errors are responsible for a large fraction of first-tapeout respins [9]. While the majority of the effort in improving design verification is dedicated to error detection, today the correction of errors is for the most part left to the skill and creativity of the designers, resulting in an extremely time-consuming activity.

As of today, a few solutions have been proposed that aim to automate the process of error correction. Some of these techniques target specific types of design errors, for instance the solutions in [2, 5, 8, 12]. Others have a broader scope but require the availability of a golden model or functional specification to isolate the location of the error [2, 4, 6, 7, 8]. Unfortunately, in many cases, such functional models are limited to a high-level description of the system, or they are only partially specified or, sometimes, completely unavailable. In this work we propose a novel technique that is suitable to correct a large number of design errors while relying exclusively on a set of simulation bug traces, hence not requiring any type of golden model of the design.

Automatic repair of design errors is especially important at the gate level because it is difficult to modify automatically-synthesized netlists manually. In some cases, it may be conceivable to fix individual design errors by repetitively synthesizing the entire netlist from scratch. However, such a design strategy is typically inefficient because (1) previously performed physical synthesis optimizations might be invalidated, and (2) it fails if the source of the error is the synthesis tool itself. Additionally, a gate-level error-correction approach offers possibilities not available when working with higher-level specifications, including reconnecting individual wires, changing individual gate types, etc.

In this work we propose a novel framework for the correction of design errors, called COunterexample-guided REsynthesis (CoRé). CoRé performs gate-level circuit repair based only on error traces composed of input vectors and output responses. Compared with other techniques, our framework presents the following advantages:

1. It does not rely on specific error models; therefore, a large number of error types can be corrected compared to previous solutions.
2. Since it only requires input vectors and output responses, it can be applied to any mainstream verification flow, as we show in Section IV-B.

Our framework derives its underlying combinational error-diagnosis technique from the work in [11], due to its flexibility and scalability, and it is powered by two innovative resynthesis techniques: *Goal-Directed Search (GDS)* and *Entropy-Guided Search (EGS)*. These techniques rely on simulation *signatures* to identify potential resynthesis options, which are then verified for correctness. If verification fails, counterexamples returned by the verification tool are used to refine error diagnosis and guide future resynthesis. Our empirical evaluation shows that this framework is able to effectively guide the resynthesis process to fix errors.

The remaining part of the paper is organized as follows. Section II introduces background and previous work. In section III we describe our resynthesis techniques in detail. Section IV discusses our CoRé framework. Experimental results are given in Section V, and section VI concludes this paper.

II. Background

In this section, we first define basic terminology and describe previous work. The error-diagnosis technique used in CoRé is introduced next, after which a commonly used error model is described as a baseline comparison to our work.

A. Terminology

When considering a digital circuit under logic simulation, we define a simulation *signature* as a bit-vector of simulated values for a wire, or internal node, where the n-th bit of the signature vector corresponds to the stable value observed on that wire for the n-th input pattern. In this paper we strive to

1-4244-0630-7/07/$20.00 ©2007 IEEE.

correct a design by re-generating the functionality of incorrect wires using other internal wires as inputs. In this context, we call the signature of the wire that needs to be resynthesized the *target signature*, and we call the signatures of wires to be considered as new inputs the *candidate signatures*.

Given a target signature s_t and a collection of candidate signatures s_{c_1}, s_{c_2},...,s_{c_n}, we say that s_t can be resynthesized by s_{c_1}, s_{c_2},...,s_{c_n} if s_t can be expressed as follows: $s_t = F(s_{c_1}, s_{c_2}, ..., s_{c_n})$, where $F(s_{c_1}, s_{c_2}, ..., s_{c_n})$ is a Boolean function called the *resynthesis function.* We also call a netlist that implements the resynthesis function the *resynthesis netlist*.

In this paper, we use $s[i]$ to denote the i-th bit of signature s. The following theorem states a necessary and sufficient condition for a resynthesis function to exist. This theorem is a special case of Theorem 4.1 in [14][1].

Theorem 1 *Consider candidate signatures s_{c_1}, s_{c_2},...,s_{c_n} and a target signature s_t. Then a resynthesis function F, where $s_t = F(s_{c_1}, s_{c_2},...,s_{c_n})$, exists if and only if no bit pair $\{i, j\}$ exists such that $s_t[i] \neq s_t[j]$ but $s_{c_k}[i] = s_{c_k}[j]$ for all $1 \leq k \leq n$.*

Based on Theorem 1, we say that a pair of bits $\{i, j\}$, where $s_t[i] \neq s_t[j]$, can be *distinguished* by signature s_{c_k} if $s_{c_k}[i] \neq s_{c_k}[j]$. We call bits $\{i, j\}$ a *Pair of Bits to be Distinguished (PBD)*. We also define the *distinguishing power* of a signature as the PBDs that can be distinguished by the signature. In other words, it is the information needed to distinguish two bits with different values in the target signature. Work based on *SPFDs* [13] also utilizes similar concepts.

B. Previous Work

Error repair involves two steps. In the first step, the circuit is diagnosed to identify potential changes that can fix the error. In the second step, the changes are implemented by resynthesis. The first step is called *Error Diagnosis* (ED), and the second step is called *Error Correction* (EC). A comparison of our work with previous error diagnosis and correction techniques is given in Table I. In the table, "Num. of errors" is the number of errors that can be corrected by the technique.

TABLE I
A COMPARISON OF ERROR DIAGNOSIS AND CORRECTION TECHNIQUES.

Technique	ED/ EC	Num. of errors	Error model	Scalability	Require-ment
ACCORD [2]	Both	Single	SLDE	Moderate (BDDs)	Func spec.
AutoFix [4]	Both	Multiple	None	Moderate (BDDs)	Golden netlist
Kuo [5]	ED	Single	Abadir	Good (ATPG)	Test vec.
Lin [6]	Both	Multiple	None	Moderate (BDDs)	Golden netlist
Madre [8]	Both	Single	PRIAM	Moderate	Func. spec.
Smith[11]	ED	Multiple	None	Good (SAT)	Test vec.
Veneris[12]	Both	Multiple	Abadir	Good (ATPG)	Test vec.
CoRé (Ours)	Both	Multiple	None	Good (SAT, signatures)	Test vec.

C. Error Diagnosis

The error-diagnosis technique used in our CoRé framework is based on the work by Smith *et al.* [11]. Given a logic netlist, a set of test vectors and a set of correct output responses, this technique will return a set of wires, also called *error sites*, along with their values for each test vector that can correct the erroneous output responses. Our CoRé framework corrects design errors by resynthesizing the error sites using the corrected values as the target signature.

D. Error Model

Error diagnosis and correction are difficult problems. As a result, error models have been introduced to classify common design errors. Abadir's model [1] is commonly used in previous work. This model includes: (1) wrong gate, (2) extra/missing wire, (3) wrong input, and (4) extra/missing gate.

As we will show in Section III, our resynthesis techniques subsume all the errors in this model. In addition to being successful in practice, CoRé is guaranteed to rectify any netlist if the test vectors are applied exhaustively. However, exhaustive enumeration of test vectors may not be practical. As a result, we assume that the discrepancy between the erroneous and the correct netlist is small, so that an acceptable fix can be found using a reasonable number of test vectors.

III. Combinational Resynthesis Techniques

Our approach to fixing errors includes two resynthesis techniques: Goal-Directed Search (GDS) and Entropy-Guided Search (EGS) The former directs the search from the target signature towards promising candidate signatures, while the latter uses entropy to guide the selection of signatures. In addition, we develop techniques to exploit full observability don't-cares during resynthesis. We first define the *entropy* of a signature, which will be used in our resynthesis techniques.

A. Entropy of a Signature

To measure the distinguishing power required by the target signature and the power possessed by a candidate signature, we define the concept of *entropy* in this subsection. To simplify bookkeeping, we order bits in the signatures so that the target signature is composed of 0s followed by 1s. Therefore, the target signature always resembles "00...0011...11".

Definition 1 *Suppose that there are x 0s and y 1s in the target signature s_t. We define the "entropy" of the signature as "$x \times y$", which is the number of Pairs of Bits to be Distinguished (PBDs) in the target signature. Given a candidate signature s_c, suppose that there are p 0s and q 1s in the first x bits, as well as r 0s and s 1s in the last y bits, we then define the "entropy of s_c with respect to s_t" as "$p \times s + q \times r$". In other words, it is the number of PBDs in the target signature that can be distinguished by the candidate signature.*

Signature	s_t	s_{c_1}	s_{c_2}	s_{c_3}	s_{c_4}
Pattern	00111	01011	10110	00101	00001

Fig. 1. Signatures and their bit patterns used in Example 1.

Example 1 *Figure 1 shows five signatures, where s_t is the target signature and s_{c_1} to s_{c_4} are candidate signatures. Signature s_t can be generated by s_{c_1}, s_{c_2} and s_{c_3} because all the PBDs in s_t can be distinguished, and the resynthesis function is $s_{c_1} \cdot s_{c_2} + s_{c_3}$. However, s_t cannot be generated by s_{c_1} and s_{c_2} because bit pair $\{5, 3\}$ cannot be distinguished. The entropy of s_t is 6, and the entropy of s_{c_1}, s_{c_2}, s_{c_3} and s_{c_4} are 3, 3, 4 and 2, respectively. It is obvious that s_t cannot be generated using*

[1] All proofs of theorems are omitted due to space limitations.

s_{c_1} and s_{c_4}, because the total number of PBDs distinguishable by s_{c_1} and s_{c_4} is smaller than the number of PBDs in s_t.

Based on Example 1, the following theorem states a necessary but not sufficient condition to determine whether the target signature can be generated from a set of candidate signatures.

Theorem 2 *Consider a target signature s_t and a set of candidate signatures $s_{c_1}...s_{c_n}$. If s_t can be generated by $s_{c_1}...s_{c_n}$, then $entropy(s_t) \leq \sum_{i=1}^{n} entropy(s_{c_i})$.*

B. Goal-Directed Search

GDS searches for resynthesis functions of the target signature using combinations of gates and candidate signatures. To reduce the search space, we devise two effective pruning techniques: the entropy test and the compatibility test. Currently, inverters and 2-input AND, OR and XOR gates are supported.

The entropy test relies on Theorem 2 to reject resynthesis opportunities when the selected candidate signatures do not have enough entropy. In other words, a collection of candidate signatures whose total entropy is less than the entropy of the target signature is not considered for resynthesis.

The compatibility test is based on the controlling values of logic gates. For example, it is impossible to generate 1 on the output of an AND gate if one of its inputs is 0. To generate 1 on the output, all its inputs must be 1. We use *compatibility constraints* to encode such criteria, which can prune the selection of inputs according to the output constraint and the gate being tried. A compatibility constraint consists of a type and a signature. Currently, three types of constraints are used.

Identity constraints specify an equivalence relationship between the input and the output of a gate. For example, the output of a buffer must be the same as its input, and the output of an inverter must be the same as the complement of its input.

Need-one constraints require that specific bits in the input signature must be 1 whenever the corresponding bits in the constraint's signature are 1. They are used to encode the constraints imposed by AND gates. Similarly, *need-zero constraints* encode the constraints imposed by OR gates.

These constraints, which propagate from the outputs of gates to their inputs during resynthesis, need to be recalculated for each gate being tried. For example, an identity constraint will become a need-one constraint after it propagates through an AND gate, and it will become a need-zero constraint if it propagates through an OR gate. The rules for calculating the constraints are shown in Figure 2, and the GDS algorithm is given in Figure 3. In the algorithm, *level* is the level of logic being explored, *constr* is the constraint and *C* returns a set of candidate resynthesis opportunities. Initially, *level* is set to 1, and *constr* is set to be identical to the target signature s_t. Function *update_constr* updates constraints according to Figure 2, and *entropy* returns the entropy of the signature or the signature of the candidate resynthesis function.

Our GDS technique subsumes the fixes required to correct the errors described in Section II-D: since we try all possible gate combinations, we will replace a wrong gate with the correct one, remove an extra gate or insert a missing gate whenever necessary; similarly, extra wire, missing wire or wrong input are handled by trying all possible wire combinations. Furthermore, we can handle errors not modeled in Section II-D, such as swap of wires or gate types.

	Identity	Need-one	Need-zero
INVERTER	Signature complemented		
BUFFER	Constraint unchanged		
AND	Need-one	Need-one	None
OR	Need-zero	None	Need-zero
XOR	None	None	None

Fig. 2. Given a constraint imposed on a gate's output and the gate type, this table calculates the constraint of the gate's inputs. The output constraints are given in the first row, the gate types are given in the first column, and their intersection is the input constraint.

```
Function GDS(level, constr, C)
 1  if (level == max_level)
 2    C= candidate signatures comply with constr
 3    return;
 4  foreach gate ∈ library
 5    constr_n= update_constr(gate, constr);
 6    GDS(level + 1, constr_n, C_n);
 7    foreach c_1, c_2 ∈ C_n
 8      if (level=1 & (entropy(c_1)+entropy(c_2))< entropy(s_t))
 9        continue;
10      Calculate signature s_n according to gate, c_1 and c_2;
11      if (s_n complies with constr)
12        C = C + gate(c_1, c_2);
```

Fig. 3. The GDS algorithm.

The major limitation of GDS is the weakening of constraints with each additional level of logic. For example, no constraints exist on the inputs of an AND gate if it is connected to the input of an OR gate. Therefore, our current implementation of GDS considers at most two levels of logic at a time, and we rely on EGS to find more complex resynthesis functions.

C. Entropy-Guided Search

EGS is based on Theorem 1, which states that a resynthesis function can be generated when a set of candidate signatures covers all the PBDs in the target signature. However, the number of collections that satisfy this criterion may be exponential. For example, any collection of signatures that involve all the primary inputs is valid. To identify possible candidate signatures effectively, we first limit our search to *m* signatures near the target wire, and then we use the following heuristics to select candidate signatures from those *m* signatures.

1. We choose *n* signatures closest to the target wire. This heuristic is based on the assumption that the discrepancy between the original netlist and the correct one is small, therefore possible signal sources for the resynthesis function should be close to the target wire.
2. We choose *o* signatures that cover the least-covered PBDs, which are the PBDs covered by a small number of signatures.
3. We select *p* signatures with high entropy. In other words, we select signatures that cover a large number of PBDs.
4. For the PBDs that remain uncovered, we cover them by selecting one signature for each PBD.
5. If the required function can not be generated using the selected signatures, we increase parameters *m*, *n*, *o* and *p*.

In our implementation, we set *m*=200, *n*=20, *o*=10 and *p*=10 initially according to our empirical observations. Although

we may select more signatures than needed for resynthesis, the logic optimizer we use in the next step is usually able to identify the redundant signatures and use only those which are essential. The input to the optimizer is the truth table of the resynthesis function, which is constructed as follows:

1. Each signature is an input to the truth table. The i-th input produces the i-th column in the table, and the j-th bit in the signature determines the value of the j-th row.
2. If the j-th bit of the target signature is 1, then the j-th row is a minterm; otherwise it is a maxterm.
3. All other terms are don't-cares.

Signature	Truth table				
s_t=0101	s_1	s_2	s_3	s_4	s_t
s_1=1010	1	0	1	0	0
s_2=0101	0	1	1	0	1
s_3=1110	1	0	1	0	0
s_4=0001	0	1	0	1	1
Minimized	0	-	-	-	1

Fig. 4. The truth table on the right is constructed from the signatures on the left. Signature s_t is the target signature, while signatures s_1 to s_4 are candidate signatures. The minimized truth table suggests that s_t can be resynthesized by an INVERTER with its input set to s_1.

Figure 4 shows an example of the constructed truth table. The truth table can be synthesized and optimized using existing synthesis software to produce a resynthesis netlist.

D. Utilizing Don't-Cares

Controllability don't-cares are used in our framework by construction. To utilize observability don't-cares, we complement the signature of the target wire and perform resimulation, which is fast because only downstream logic will be resimulated. If the change of the signature does not propagate to any primary output, it is a don't-care and is not considered during resynthesis. Since we observe the changes at primary outputs, complete observability don't-cares are obtained.

IV. Error-Correction Framework

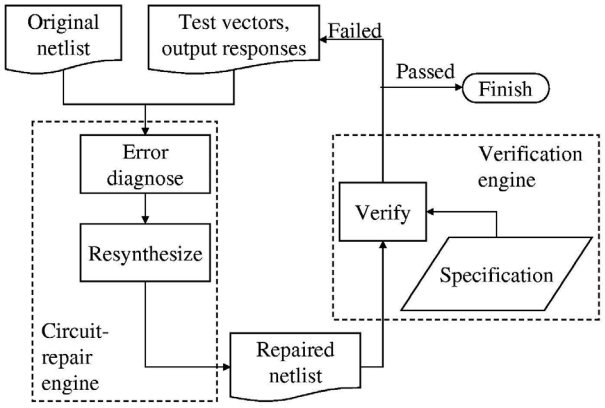

Fig. 5. Flow chart of the CoRé framework.

In our error-correction framework, CoRé, the netlist produced by the circuit-repair engine is checked by a verification engine, and the results are used to refine error diagnosis and guide resynthesis. Since the error-diagnosis technique we adopted supports combinational logic only, we assume that the state values are known when correcting errors in sequential circuits. This framework is outlined in Figure 5 and will be discussed in detail below.

A. The CoRé Framework

In CoRé framework, a test vector is called a *preserving vector* if its output responses are correct, and the vector is called an *error-sensitizing vector* if its output responses are incorrect. Error-sensitizing vectors are often called *counterexamples*.

```
framework CoRé(CKT, vectors_p, vectors_e, CKT_n)
 1   calculate CKT's initial signatures using vectors_p and
     vectors_e;
 2   diagnose CKT to generate fixes using vectors_e;
 3   foreach fix ∈ fixes
 4     produce CKT_n by resynthesizing fix;
 5     counterexample=verify(CKT_n);
 6     if (counterexample is empty)
 7       return CKT_n;
 8     else
 9       if (verify(CKT, counterexample) failed)
10         rediagnose CKT using counterexample and update
           fixes;
11     simulate counterexample and update CKT's signatures;
```

Fig. 6. The CoRé framework.

Our CoRé framework is described in Figure 6. The inputs to the framework are the original netlist (CKT), the initial preserving vectors ($vectors_p$) and the initial error-sensitizing vectors ($vectors_e$). The output is the rectified netlist CKT_n. The framework first uses the error-diagnosis technique described in Section II-C to generate potential fixes. Each fix consists of one or more error sites along with their signatures. The bits in those signatures that correspond to the error-sensitizing vectors are corrected according to the diagnosis results, while the bits that correspond to the preserving vectors remain unchanged. These fixes are then implemented by our resynthesis techniques to produce a new circuit CKT_n as shown in lines 3-4. CKT_n is then checked by the verification engine. If verification fails, error-sensitizing vectors for CKT_n will be returned in $counterexample$. If no such vectors exist, the circuit is rectified successfully and CKT_n will be returned as shown in lines 5-7. If such vectors exist, CKT_n is abandoned, and CKT is verified against $counterexample$. If verification fails, the vectors are considered error-sensitizing and are used to refine error diagnosis; otherwise they are preserving vectors. These vectors are then simulated to update CKT's signatures so that the same resynthesis attempt will not be tried again. By considering both preserving and error-sensitizing vectors, our corrections will not introduce new errors.

B. Applications

We now develop applications of our techniques in three different verification contexts.

Application 1: combinational equivalence checking and enforcement. This application fixes an erroneous netlist so that it becomes equivalent to a golden netlist. In this application, the verification engine is an equivalence checker. Test vectors on which the erroneous circuit and the golden model agree are preserving vectors, and the remaining test vectors are error-sensitizing. Initial vectors can be obtained by random simulation or equivalence checking.

Application 2: fixing errors found by simulation. This application corrects design errors that break a regression test.

TABLE II
ERROR-CORRECTION EXPERIMENT. THE BENCHMARKS IN THE TOP-HALF COMPLY WITH ABADIR'S ERROR MODE, WHILE THE BOTTOM-HALF DO NOT. NUMBER OF ITERATIONS IS THE NUMBER OF ERROR-CORRECTION ATTEMPTS PROCESSED BY THE VERIFICATION ENGINE.

Benchmark	Gate count	Type of injected error	GDS				EGS			
			Runtime (sec)			Number of iterations	Runtime (sec)			Number of iterations
			Error correction	Error diagnosis	Verification		Error correction	Error diagnosis	Verification	
S1488	636	Single gate change	1	3	1	1	1	4	1	1
S15850	685	Connection change	1	5	1	2	2	5	1	1
S9234_1	974	Single gate change	1	10	1	1	1	9	1	1
S13207	1219	Connection change	1	5	1	1	1	5	1	1
S38584	6727	Single gate change	1	306	83	1	1	306	81	1
S838_1	367	Multiple gate changes	N/A				1	6	1	1
S13207	1219	Multiple missing gates	N/A				3	12	3	6
AC97_CTRL	11855	Multiple connection changes	N/A				2	1032	252	5

TABLE III
ERROR-CORRECTION EXPERIMENT. THE NUMBER OF INITIAL PATTERNS IS REDUCED TO 64 IN THIS EXPERIMENT TO MIMIC DIFFICULT ERRORS.

Benchmark	Gate count	Type of injected error	GDS				EGS			
			Runtime (sec)			Number of iterations	Runtime (sec)			Number of iterations
			Error correction	Error diagnosis	Verification		Error correction	Error diagnosis	Verification	
S1488	636	Single gate change	1	5	3	13	1	4	1	3
S15850	685	Connection change	1	3	1	5	53	4	5	42
S9234_1	974	Single gate change	1	8	3	6	1	10	3	4

In this application, the verification engine is the simulator and the regression suite. Test vectors that break the regression are error-sensitizing vectors, and all other vectors are preserving vectors. Initial vectors can be obtained by collecting the inputs applied to the netlist while running the regression.

Application 3: fixing errors found by formal verification. This application assumes that a formal tool proves that a property can be violated, and the goal is to fix the netlist to prevent the property from being violated. In this application, counterexamples returned by the tool are error-sensitizing vectors.

V. EXPERIMENTAL RESULTS

We implemented our CoRé framework using the OAGear package [16] because it provides convenient logic representations for circuits. We adopted Smith's [11] algorithm and integrated MiniSAT [3] into our system for error diagnosis and equivalence checking. We use Espresso [10] to optimize the truth table returned by EGS, and then we construct the resynthesis netlist using AND, OR and NOT gates. Our testcases are selected from IWLS2005 benchmarks [15] based on designs from ISCAS89 and OpenCores suites. In our implementation, we limit the number of attempts to resynthesize a wire to 30, and we prioritize our correction by starting from wires closer to primary inputs. We conducted three experiments on a 2.0GHz Pentium 4 workstation. The first two experiments are in the context of equivalence checking, and the third one deals with simulation-based verification.

Equivalence checking: our first experiment employs Application 1 described in Section IV-B to repair an erroneous netlist by enforcing equivalency. Inputs and outputs of the sequential elements in the benchmarks are treated as primary outputs and inputs, respectively. The initial vectors are obtained by simulating 1024 random patterns. One error is injected to each netlist. In the first half of the experiment, the injected errors fit in the error model described in Section II-D; while the errors injected in the second half involve more than 2 levels of logic and do not comply with the error model. We apply GDS and EGS separately to compare their error-correction power and performance. Since GDS subsumes existing techniques that are based on error models, it can be used as a comparison to them. The results are summarized in Table II. As expected, GDS cannot repair netlists in the second half of the experiment, showing that our resynthesis techniques can fix more errors than those based on Abadir's error models [5, 12].

From the results in the first half, we observe that both GDS and EGS perform well in the experiment: the resynthesis time is short, and the number of iterations is typically small. This result shows that the error-diagnosis technique we adopted is effective and our resynthesis techniques repair the netlists correctly. Compared with the error-correction time required by some previous techniques that enumerate possible fixes in the error model [2, 12], the short runtime of GDS shows that our pruning methods are efficient, even though GDS also explores all possible combinations. We observe that the program runtime is dominated by error diagnosis and verification, which highlights the importance of developing faster error-diagnosis and verification techniques.

Errors that are difficult to diagnose and correct often need additional test vectors and iterations. In order to evaluate our techniques on fixing difficult errors, we reran the first three benchmarks and reduced the number of their initial patterns to 64. The results are summarized in Table III, where the number of iterations increased as expected. The results suggest that our techniques continue to be effective for difficult errors, where all the errors can be fixed within two minutes. We also observe that EGS may sometimes need more iterations due to its much larger search space. However, our framework will guide both techniques to the correct fix eventually.

In our second experiment, we inject more than one error into the netlist. The injected errors comply with Abadir's model and can be fixed by both GDS and EGS. To mimic difficult errors, the number of initial vectors is 64. We first measure the runtime and the number of iterations required to fix each error separately, we then show the results on fixing multiple errors. Time-out is set to 30 minutes in this experiment, and

TABLE IV
MULTIPLE ERROR EXPERIMENT. TIME-OUT IS SET TO 30 MINUTES AND IS MARKED AS T/O IN THE TABLE.

Benchmark	Runtime (sec)					Number of iterations				
	Error1	Error2	Error3	Error1+2	Error1+2+3	Error1	Error2	Error3	Error1+2	Error1+2+3
S1488 (GDS)	4	6	4	10	t/o	8	5	2	22	t/o
S1488 (EGS)	14	5	5	34	9	32	4	2	45	14
S13207 (GDS)	10	10	6	12	75	11	5	1	10	19
S13207 (EGS)	7	9	6	14	74	4	5	1	16	15
S15850 (GDS)	4	3	4	5	7	1	1	1	1	1
S15850 (EGS)	4	3	5	5	10	1	1	13	1	11

the results are summarized in Table IV. Similar to other error diagnosis and correction techniques, runtime of our techniques grows significantly with each additional error. However, we can observe from the results that the number of iterations is usually smaller than the product of the number of iterations for each error. It shows that our framework tends to guide the resynthesis process to fix the errors instead of merely trying all possible combinations of fixes. Another interesting phenomenon is that EGS can simultaneously fix all three errors in the S1488 benchmark, while GDS cannot. The reason is that EGS found a fix involving only two wires even though three errors were injected. Since GDS could not fix the netlist using only two error sites, three-error diagnosis was performed, which was extremely slow. The reason is that in addition to fixes involving three error sites, any combination of wires consisting of two error sites and one "healthy" site (site with its function unchanged) is also a valid fix. As a result, the number of possible fixes increased dramatically and evaluating all of them was time consuming. This explanation is confirmed by the following observation: error diagnosis returned 8, 7 and 9 possible fixes for error1, error2 and error3 respectively, while the number of fixes for all three errors using three sites was 21,842. This situation suggests that EGS is more powerful than GDS, as well as many techniques subsumed by GDS.

TABLE V
ERROR CORRECTION IN THE CONTEXT OF SIMULATION-BASED VERIFICATION. WE SIMULATE 1024 PRESERVING AND m ERROR-SENSITIZING VECTORS, WHERE THE ERROR-SENSITIZING VECTORS RANDOMLY CHANGE ONE OUTPUT PER VECTOR.

Bench-mark	Runtime (sec)				Number of error sites			
	m=1	m=2	m=3	m=4	m=1	m=2	m=3	m=4
S1488	3	4	10	10	1	2	3	3
S15850	3	4	4	6	1	2	2	4
S13207	3	6	8	19	1	2	3	5

Simulation-based verification: in our third experiment, we simulate n preserving vectors and m error-sensitizing vectors, where $m << n$. Error-sensitizing vectors are produced by randomly changing one output per vector. We then check whether our framework can produce a netlist that is adaptive to the new responses. This is similar to fixing errors found by simulation-based verification, where a few vectors break the regression test while most vectors should be preserved. In this experiment, we set n=1024 while changing m, and the results are summarized in Table V. We can observe from the results that additional error-sensitizing vectors usually require more wires to be fixed, and the runtime is also longer. However, our framework is able to repair all the benchmarks within a short time by resynthesizing only a small number of wires. This result suggests that our framework works effectively in the context of simulation-based verification.

VI. CONCLUSIONS

In this paper we proposed the CoRé framework to correct functional errors in combinational circuits using only error traces. This framework is powered by two innovative resynthesis techniques, GDS and EGS, also presented in this paper. Because CoRé does not rely on specific error models, it offers more error-correction capabilities than many previous solutions. The experimental results show that our framework corrects more error types than solutions based on specific error models, and provides better performance. Furthermore, our experiments in the context of simulation-based verification show that CoRé can always produce a rectified netlist which eliminates the erroneous responses while maintaining the correct ones. In addition, CoRé can be easily adopted in most verification flows. We are currently work on extending CoRé to fix errors in sequential circuits.

REFERENCES

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

[2] P.-Y. Chung and I. N. Hajj, "ACCORD: Automatic Catching and CORrection of Logic Design Errors in Combinational Circuits", *ITC'92*, pp. 742-751.

[3] N. Eén and N. Sörensson, "An Extensible SAT-solver", *Theory and Applications of Satisfiability Testing, SAT*, 2003, pp. 502-518.

[4] S.-Y. Huang, K.-C. Chen and K.-T. Cheng, "AutoFix: A Hybrid Tool for Automatic Logic Rectification", *IEEE TCAD*, pp. 1376-1384, Sep. 1999.

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

[6] C.-C. Lin, K.-C. Chen and M. Marek-Sadowska, "Logic Synthesis for Engineering Change", *IEEE TCAD*, pp.282-202, Mar. 1999.

[7] C.-H. Lin et al., "Design and Design Automation of Rectification Logic for Engineering Change", *ASPDAC'05*, pp. 1006-1009.

[8] 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.

[9] Prakash Rashinkar, Peter Paterson, and Leena Singh. *System-on-a-chip Verification: Methodology and Techniques*. Kluwer Academic Publishers, 2002.

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

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

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

[13] S. Yamashita, H. Sawada and A. Nagoya, "SPFD: A new method to express functional flexibility", *IEEE TCAD*, pp. 840-849, Aug. 2000.

[14] J. Zhang, S. Sinha, A. Mishchenko, R. Brayton and M. Chrzanowska-Jeske, "Simulation and Satisfiability in Logic Synthesis", *IWLS '05*, pp. 161-168.

[15] `http://iwls.org/iwls2005/benchmarks.html`

[16] `http://www.openedatools.org/projects/oagear/`