Fixing Design Errors with Counterexamples and Resynthesis

Kai-hui Chang, Igor L. Markov and Valeria Bertacco
University of Michigan at Ann Arbor

Abstract— **In this work we propose a resynthesis framework, called CoRé, that automatically corrects errors in digital designs. The framework is based on a simulation-based abstraction technique and performs error correction through two innovative circuit resynthesis solutions: Distinguishing-Power Search (DPS) and Goal-Directed Search (GDS), which modify the functionality of a circuit's internal nodes to match the correct behavior. In addition, we propose a compact encoding of resynthesis information, called Pairs of Bits to be Distinguished (PBDs), which is a key enabler for our resynthesis techniques. Compared with previous solutions, CoRé is more powerful in that: (1) it can fix a broader range of error types because it is not bounded by specific error models; (2) it derives the correct functionality from simulation vectors, without requiring golden netlists; and (3) it can be applied with a broad range of verification flows, including formal and simulation-based.**

Index Terms— **Error correction, error diagnosis, logic synthesis**

I. Introduction

Due to the dramatic increase in design complexity of modern electronics, digital systems are often released with many latent errors, some of which have the potential of triggering expensive damage or replacement costs. While recent improvements in verification enable engineers to more efficiently expose a larger fraction of design errors, little effort has been devoted to automatically fixing such errors. As a result, existing techniques in this domain have very limited power and scalability.

The process of repairing functional design bugs involves two steps: error diagnosis and error correction. Error diagnosis identifies the portion of the design responsible for the error, while error correction is responsible for locally modifying the functionality of the identified portion through a specialized synthesis process, called *resynthesis*. Recent work by Smith *et al.* [6] and Ali *et al.* [1] greatly improved the scalability and efficiency of error diagnosis. However, fixing errors via resynthesis remains challenging because existing techniques lack the scalability to handle the global implications of the logic modifications imposed by error correction. As a result, state-of-art techniques often limit the types of errors that can be corrected [8] or operate only on small circuits [7], [10].

In this work we present an innovative framework, called *COunterexample-guided REsynthesis (CoRé)*, which can overcome the limitations discussed above, while also admitting a simple and particularly efficient implementation. It addresses automatic error correction for a broad range of design errors and on larger scale designs, both combinational and sequential. CoRé is based on a simulation-driven abstraction technique and requires only input stimuli and correct output responses (no simulation values at internal nodes) to perform its analysis and suggest a design fix. Because of these simple requirements, it can be applied to a variety of verification methodologies, including formal and simulation-based flows. The CoRé framework is composed of two main engines to perform resynthesis, called *Distinguishing-Power Search (DPS)* and *Goal-Directed Search (GDS)*. In addition, the internal analysis of CoRé relies on a compact encoding of resynthesis information, called *Pairs of Bits to be Distinguished (PBDs)*, which is based on signatures and enables efficient computation of Don't-Cares (DCs). PBDs are pivotal to the scalability of CoRé and allow CoRé to generate a broad variety of resynthesis solutions.

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

II. Background

In this work we assume that an input design, with one or more bugs, is provided as a Boolean network. We strive to correct its erroneous behavior by regenerating the functionality of incorrect nodes. This section starts by defining some terminology and then overviews relevant previous work.

A. Signatures and Distinguishing Power

Definition 1: Given a node t in a Boolean network, whose function is f, and input vectors x_1, x_2 ... x_k, we define the *signature* of node t, s_t, as $(f(x_1),...,f(x_k))$, where $f(x_i) \in \{0,1\}$ represents the output of f given an input vector x_i.

Our goal is to modify the functions of the nodes responsible for the erroneous behavior of a circuit via resynthesis. In this context, we call a node to be resynthesized the *target node*, and we call the nodes that we can use as inputs to the newly synthesized node (function) the candidate nodes. Their corresponding signatures are called the *target signature* and the *candidate signatures*, respectively.

Given a target signature s_t and a collection of input 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 $s_t = f(s_{c_1}, s_{c_2}, ..., s_{c_n})$, where $f(s_{c_1}, s_{c_2}, ..., s_{c_n})$ is a vector 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 proposition below states that a sufficient and necessary condition for a resynthesis function to exist is that, whenever two bits in the target signature are distinct, then such bits need to be distinct in at least one of the candidate signatures.[1]

Proposition 1: Consider a collection of 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 $1 \leq k \leq n$.

In this work we call a pair of bits $\{i,j\}$ in s_t, where $s_t[i] \neq s_t[j]$, a *Pair of Bits to be Distinguished (PBD)*. Based on Prop. 1, we say that the PBD $\{i,j\}$ can be *distinguished* by signature s_{c_k} if $s_{c_k}[i] \neq s_{c_k}[j]$. We define the *Required Distinguishing Power (RDP)* of the target signature s_t, $RDP(s_t)$, as the set of PBDs that need to be distinguished. We also define the

This work was partially funded by the NSF under Award 0448189.

[1]This proposition is a special case of Theorem 4.1 in [11], where the minterms appearing in signatures represent the care-terms and all other minterms are DCs.

Distinguishing Power (DP) of a candidate signature s_{c_k} with respect to the target signature s_t, $DP(s_{c_k}, s_t)$, as the set of PBDs in s_t that can be distinguished by s_{c_k}. With this definition, Prop. 1 can be restated as "a resynthesis function, f, exists if and only if $RDP(s_t) \subseteq \cup_{k=1}^{n} DP(s_{c_k}, s_t)$".

B. Don't-Cares

When considering a sub-network within a large Boolean network, Don't-Cares (DCs) are exploited by many synthesis techniques because they provide additional freedom for optimizations. *Satisfiability Don't-Cares (SDCs)* occur when certain combinations of input values do not occur for the sub-network, while *Observability Don't-Cares (ODCs)* occur when the output values of the sub-network do not affect any primary output. As we show in Section III-A, our CoRé framework is able to utilize both SDCs and ODCs.

C. Related Work

Existing error repair techniques often partition the problem into error diagnosis and error correction. A comparison of error diagnosis and correction techniques can be found in Table I of our preliminary work [2], which was limited to the analysis of combinational circuits.

The error-diagnosis portion of the CoRé framework is based on the work by Smith *et al.* [6]. The diagnosis technique considers a Boolean network, a set of input test vectors and a set of correct output responses. For each input test vector, it will return a set of nodes (called *error sites*) found to compute incorrect values along with the corresponding correct values. The correction portion of the CoRé framework then corrects design errors by resynthesizing the error sites with functions that generate the proper correct values for each input vector.

III. Error-Correction Framework

For the discussion in Sections III-A and III-B we restrict our analysis to combinational designs. In this context, the correctness of a circuit is simply determined by the output responses under all possible input vectors. We will show in Section III-C how to extend the solution to sequential designs.

CoRé, our error-correction framework, relies on simulation to generate signatures, which constitute our abstract model of the design and are the starting point for the error diagnosis and resynthesis algorithms. After the netlist is repaired, it is checked by a verification engine. If verification fails, possibly due to new errors introduced by the correction process, new counterexamples are generated and used to further refine the abstraction. Although in our implementation we adopted Smith's error diagnosis technique [6] due to its scalability, alternative diagnosis techniques can be used as well.

A. The CoRé Framework

In CoRé, an input test vector is called a *functionality-preserving vector* if its output responses comply with the specification, and the vector is called an *error-sensitizing vector* if its output responses differ. *Error-sensitizing vectors* are often called *counterexamples*.

The algorithmic flow of CoRé is outlined in Figure 1. The inputs to the framework are the original buggy netlist (CKT_{err}), the initial *functionality-preserving vectors* ($vectors_p$) and the initial *error-sensitizing vectors* ($vectors_e$). The output is the rectified netlist CKT_{new}. The framework first performs error diagnosis to identify error locations and the correct values that should be generated for those locations so that the *error-sensitizing vectors* could produce the correct output responses. Those error locations constitute the *target nodes* for resynthesis. The bits in the target nodes' signatures that correspond to the *error-sensitizing vectors* must be corrected according to the diagnosis results, while the bits that correspond to the *functionality-preserving vectors* must remain unchanged. If we could somehow create new combinational netlist blocks that generate the required signatures at the target nodes using other nodes in the Boolean network, we would be able to correct the circuit's errors, at least those that have been exposed by the *error-sensitizing vectors*. Let us assume for now that we can create such netlists (techniques to this end will be discussed in the next section), producing the new circuit CKT_{new} (line 4). CKT_{new} is checked at line 5 using the verification engine. When verification fails, new *error-sensitizing vectors* for CKT_{new} will be returned in *counterexample*. If no such vector exists, the circuit has been successfully corrected and CKT_{new} is returned. Otherwise, CKT_{new} is abandoned, while *counterexample* is classified either as *error-sensitizing* or *functionality-preserving* with respect to the original design (CKT_{err}). If *counterexample* is *error-sensitizing*, it will be added to $vectors_e$ and used to rediagnose the design. CKT_{err}'s signatures are then updated using *counterexample*. By accumulating both *functionality-preserving* and *error-sensitizing vectors*, CoRé will avoid reproposing the same wrong correction; hence guaranteeing that the algorithm will eventually complete. Figure 2 illustrates a possible execution scenario with the flow just described.

```
CoRé($CKT_{err}, vectors_p, vectors_e, CKT_{new}$)
 1  $compute\_signatures(CKT_{err}, vectors_p, vectors_e)$;
 2  $fixes$= $diagnose(CKT_{err}, vectors_e)$;
 3  foreach $fix \in fixes$
 4      $CKT_{new}$= $resynthesize(CKT_{err}, fix)$;
 5      $counterexample$=$verify(CKT_{new})$;
 6      if ($counterexample$ is empty)   return $CKT_{new}$;
 7      else if ($counterexample$ is error-sensitizing for $CKT_{err}$)
 8          $vectors_e = vectors_e \cup counterexample$;
 9          $fixes$= $rediagnose(CKT_{err}, vectors_e)$;
10      $update\_signatures(CKT_{err}, counterexample)$;
```

Fig. 1. The algorithmic flow of CoRé.

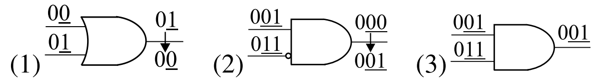

Fig. 2. Execution example of CoRé. Signatures are shown above the wires, where underlined bits correspond to *error-sensitizing vectors*. (1) The gate was meant to be AND but is erroneously an OR. Error diagnosis finds that the output of the 2nd pattern should be 0 instead of 1; (2) the first resynthesis netlist fixes the 2nd pattern, but fails further verification (the output of the 3rd pattern should be 1); (3) the counterexample from step 2 refines the signatures, and a resynthesized netlist that fixes all the test patterns is found.

SDCs are exploited in CoRé by construction because simulation can only produce legal signatures. To utilize ODCs, we simulate the complement signature of the target node and mark the bit positions whose changes do not propagate to any primary output as ODCs: those positions are not considered during resynthesis. Note that if a diagnosis contains multiple error sites, the sites that are closer to primary outputs should be resynthesized first so that the downstream logic of a node is always known when ODCs are calculated.

B. Analysis of the Framework

CoRé is more effective than many previous solutions because it supports the use of SDCs and ODCs, including external DCs. External SDCs can be exploited by providing only legal input patterns when generating signatures, while external ODCs are utilized by marking uninterested output vectors don't-cares.

To achieve the required scalability to support the global implications of error correction, CoRé uses an abstraction-refinement scheme: signatures provide an abstraction of the Boolean network for resynthesis because they are the nodes' partial truth tables (all unseen input vectors are considered as DCs), and the abstraction is refined by means of the counterexamples that fail verification. The following proposition shows that CoRé can eventually always produce a netlist which passes verification. However, as it is the case for most techniques based on abstraction and refinement, the framework may time-out before a valid correction is found. The use of high-quality test vectors [8] is effective in alleviating this potential problem.

Proposition 2: Given a buggy combinational design and a specification that defines the output responses of each input vector, the CoRé algorithm can always generate a netlist that produces the correct output responses.

Proof: Given a set of required "fixes", the resynthesis functions of CoRé can always generate a correct set of signatures, which in turn produce correct responses at primary outputs. Observe that each signature represents a fragment of a signal's truth table. Therefore, when all possible input patterns are applied to our CoRé framework, the signatures essentially become complete truth tables, and hence define all the terms required to generate correct output responses for any possible input stimulus. In CoRé, all the counterexamples that fail verification are used to expand and enhance the set of signatures. Each correction step of CoRé guarantees that the output responses of the input patterns seen so far are correct, thus any counterexample must be new. However, since the number of distinct input patterns is finite, eventually no new vector can be generated, guaranteeing that the algorithm will complete in a finite number of iterations. In practice, we find that a correct design can often be found in a few iterations. ■

C. Sequential Circuits

The discussion so far has addressed only combinational circuits. CoRé is easily adaptable to correct sequential circuits, too, as described in this section. First of all, when operating on sequential circuits the user will provide CoRé with input traces, instead of input patterns. A trace is a sequence of input patterns, where a new pattern is applied to the design's inputs at each simulation cycle, and the trace can be either *error-sensitizing* or *functionality-preserving*. To address sequential circuits, we adopt the diagnosis techniques from Ali *et al.* [1] relating to sequential circuits. The idea is to first unroll the circuit by connecting the outputs of the state registers to the inputs of the registers in the previous cycle, and then use the test vectors to constrain the unrolled circuit. Given an initial state and a set of test vectors with corresponding correct output responses, Ali's error-diagnosis technique is able to produce a collection of error sites, along with their correct values, that rectify the incorrect output responses.

To correct sequential designs we apply the same algorithm described in Section III-A with two changes: the diagnosis procedure should be as described in [1], and the signature generation function is modified so that it can be used in a sequential design. Specifically, the new sequential signature generation procedure should record one bit of signature for each cycle of each sequential trace that we simulate. For instance, if we have two traces available, a 4-cycle trace and a 3-cycle trace, we will obtain a 7-bit signature at each internal circuit node. An example of the modified signature is shown in Figure 3. In our current implementation, we only use combinational ODCs. In other words, we still treat inputs of state registers as primary outputs when calculating ODCs. Although it is possible to exploit sequential ODCs for resynthesis, we do not pursue this optimization, yet.

	Trace1				Trace2		
Cycle	1	2	3	4	1	2	3
Signature	0	1	1	0	1	0	1

Fig. 3. Sequential signature construction example. The signature of a node is built by concatenating the simulated values of each cycle for all the bug traces. In this example, trace1 is 4 cycles and trace2 is 3 cycles long. The final signature is then 0110101.

IV. Resynthesis Techniques

The basis for CoRé's resynthesis solution is the signature available at each internal circuit node. The resynthesis problem is formulated as follows: given a target signature, find a resynthesis netlist that generates the target signature using the signatures of other nodes in the Boolean network as inputs. In this section, we first define the *absolute distinguishing power* $|DP(s)|$ of a signature s, and then we propose a Distinguishing-Power Search (DPS) technique that uses $|\mathrm{DP}|$ to select candidate signatures and generates the required resynthesis netlist. Next, we propose a Goal-Directed Search (GDS) technique that can find a resynthesis netlist with the smallest possible logic depth. The ability to generate a minimum-depth netlist is important if circuit timing is a concern. Finally, we briefly compare our approach to other techniques.

A. Absolute Distinguishing Power of a Signature

In this subsection we define the concept of *absolute distinguishing power*, which provides search guiding and pruning criteria for our resynthesis techniques. To simplify bookkeeping, we reorder bits in every signature so that in the target signature all the bits with value 0 precede the ones with value 1, as in "00...0011...11".

Definition 2: Assume a target signature s_t is composed of x 0s followed by y 1s, we define the *absolute required distinguishing power* of s_t as $|RDP(s_t)| = xy$, which is the number of PBDs in s_t. Moreover, if a candidate signature s_c has p 0s and q 1s in its first x bit positions, and r 0s and s 1s in the remaining y positions, then we define the *absolute distinguish power* of s_c with respect to s_t as $|DP(s_c, s_t)| = ps + qr$, which is the number of PBDs in s_t that can be distinguished by s_c.

The following corollary states a necessary but not sufficient condition to determine whether the target signature can be generated from a collection of candidate signatures.

Corollary 1: Consider a target signature s_t and a collection of candidate signatures $s_{c_1}...s_{c_n}$. If s_t can be generated by $s_{c_1}...s_{c_n}$, then $|RDP(s_t)| \leq \sum_{i=1}^{n} |DP(s_{c_i}, s_t)|$.

B. Distinguishing-Power Search

Distinguishing-Power Search (DPS) is based on Prop. 1, which states that a resynthesis function can be generated when a collection of candidate signatures covers all the PBDs in the target signature. However, the number of collections satisfying this criterion may be exponential. To identify possible candidate signatures effectively, we first select signatures that cover the least-covered PBDs, second those that have high |DP| (i.e., signatures that cover the most number of PBDs), and third those that cover any remaining uncovered PBD. For efficiency, we limit the search pool to the 200 nodes which are topologically closest to the target node; however, we may go past this limit when those are not sufficient to cover all the PBDs in the target signature. Finally, we exclude from the pool those nodes that are in the fanout cone of the target node, so that we avoid creating a combinational loop inadvertently.

After the candidate signatures are selected, a truth table for the resynthesis function is built from the signatures (detailed steps can be found in [2, Section III-C]). The truth table can be synthesized and optimized using existing software, such as Espresso [4] or MVSIS [12]. Note that our resynthesis technique does not require that the support of the target function is known *a priori*, since the correct support will be automatically selected when DPS searches for a set of candidate signatures that distinguishes all the PBDs. This is in contrast with other previous solutions which require that the support of the target node be known before attempting to synthesize the function.

C. Goal-Directed Search

GDS performs an exhaustive search for resynthesis netlists. To reduce the search space, we propose two pruning techniques: the |DP| test and the compatibility test. Currently, BUFFERs, INVERTERs, and 2-input AND, OR and XOR gates are supported.

The |DP| test relies on Corollary 1 to reject resynthesis opportunities when the selected candidate signatures do not have sufficient |DP|. In other words, a collection of candidate signatures whose total |DP| is less than the |RDP| of the target signature is not considered for resynthesis.

The compatibility test is based on the controlling values of logic gates. To utilize this feature, we propose three rules, called *compatibility constraints*, to prune the selection of inputs according to the output constraint and the gate being tried. Each constraint is accompanied with a signature. In particular, an *identity constraint* requires the input signature to be identical to the constraint's signature; and a *need-one constraint* requires that specific bits in the input signatures must be 1 whenever the corresponding bits in the constraint's signature are 1. *Identity constraints* are used to encode the constraints imposed by BUFFERs and INVERTERs, while *need-one constraints* are used by AND gates. Similarly, *need-zero constraints* are used by OR gates. For example, if the target signature is 0011, and the gate being tried is AND, then the *need-one constraint* will be used. This constraint will reject signature 0000 as the gate's input because its last two bits are not 1, but it will accept 0111 because its last two bits are 1. 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* when it propagates through an AND gate, and it will become a *need-zero constraint* when it propagates through an OR gate. The rules for calculating the constraints are shown in Figure 4.

	Identity	Need-one	Need-zero
INVERTER	S.C.	S.C.+Need-zero	S.C.+Need-one
BUFFER	Constraint unchanged		
AND	Need-one	Need-one	None
OR	Need-zero	None	Need-zero

Fig. 4. 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. "S.C." means "signature complemented."

The GDS algorithm is given in Figure 5. In the algorithm, *level* is the level of logic being explored, *constr* is the constraint, and C returns a set of candidate resynthesis netlists. Initially, *level* is set to 1, and *constr* is *identity constraint* with signature equal to the target signature s_t. Function *update_constr* is used to update constraints.

```
Function $GDS(level, constr, C)$
1  if ($level == max\_level$)
2    $C$= candidate nodes whose signatures comply with $constr$;
3    return;
4  foreach $gate \in library$
5    $constr_n$= $update\_constr(gate, constr)$;
6    $GDS(level+1, constr_n, C_n)$;
7    foreach $c_1, c_2 \in C_n$
8      if ($level >$ 1 or $|DP(c_1,s_t)|$+$|DP(c_2,s_t)|$$\geq$$|RDP(s_t)|$)
9        $s_n$ = $calculate\_signature(gate, c_1, c_2)$;
10       if ($s_n$ complies with $constr$)
11         $C = C \cup gate(c_1, c_2)$;
```

Fig. 5. The GDS algorithm.

GDS can be used to find a resynthesis netlist with minimal logic depth. This is achieved by calling GDS iteratively, with an increasing value of the *level* parameter, until a resynthesis netlist is found. However, the pruning constraints weaken with each additional level of logic in GDS. Therefore, the maximum logic depth for GDS is typically small, and we rely on DPS to find more complex resynthesis functions.

D. Discussion

Our use of PBDs is related to the SPFD technique used by several groups previously [5], [9], [10], where SPFD is a representation of Boolean functions that allows the use of DCs during synthesis. In particular, the approximate-SPFD technique proposed recently by Yang *et al.* [10] is somewhat similar to our approach because PBDs compactly encode a subset of the bipartite SPFD graph. However, based on the data in [10] and our experiments on ISCAS'85 benchmarks, we conservatively estimate that our techniques are at least twice as fast as those in [10] (details not reported here due to page limitations). In addition, PBDs should be more memory-efficient because they are calculated using signatures.

Several existing techniques, such as [8], also use simulation to identify potential error-correction options and rely on

TABLE I

ERROR-REPAIR RESULTS FOR SEQUENTIAL CIRCUITS. DPS IS USED IN THIS EXPERIMENT. THE ERROR DIAGNOSIS TECHNIQUE IS BASED ON [1]

Benchmark	Description	#Cells	Bug description	Err. diag. time (sec)		#Fixes	Resynthesis netlist		DPS time (sec)
				1st	Total		#Supports (min/max)	#Gates (min/max)	
Pre_norm	Part of FPU	1877	8-bit reduced OR $\rightarrow$ AND	29.4	50.8	1	19/19	83/83	0.4
MD5	MD5 full chip	13111	Incorrect state transition	5294	5670	2	33/64	58/126	28.2
DLX	5-stage pipeline MIPS-Lite CPU	14725	JAL inst. leads to incorrect bypass from MEM stage	25674	78834	54	1/21	1/944	1745
			Incorrect inst. forwarding	29436	30213	6	1/2	1/2	85

further simulation to prune unpromising candidates. Compared with these techniques, our framework is more flexible because it performs abstraction and refinement on the design itself. As a result, this framework can easily adopt new error diagnosis or correction techniques. For example, our error-correction engine can be easily replaced by any synthesis tool that can handle truth tables or cubes. Most existing techniques, however, do not have this flexibility.

V. EXPERIMENTAL RESULTS

In [2] we have shown that CoRé can effectively correct errors in combinational circuits. In this work we apply CoRé to repair errors in sequential circuits using techniques described in Section III-C. Due to space limitations, we only report the results using DPS. Note that diagnosing errors in sequential circuits is much more difficult than that in combinational circuits because circuit unrolling is used. For example, the bug trace for the last benchmark has 77 cycles, and it produces an unrolled circuit containing more than one million standard cells. The characteristics of the benchmarks and their results are summarized in Table I. For each benchmark, 32 traces were provided, and the goal was to repair the circuit so that it produces the correct output responses for those traces. Since our algorithm processes all the traces simultaneously, only one iteration will be required. For the computation of more representative runtimes only, we deliberately processed the traces one by one and failed all verification so that all the benchmarks underwent 32 iterations. All the bugs were injected at the RTL, and the designs were synthesized using Cadence RTL compiler 4.10. In the table, "Err. Diag. time" is the time spent on error diagnosis, "#Fixes" is the number of valid fixes returned by CoRé, and "DPS time" is the runtime of DPS. The minimum/maximum numbers of support variables and gates used in the returned fixes are shown under "Resynthesis netlist". Note that implementing any valid fix is sufficient to correct the circuit's behavior, and we rank the fixes based on the logic depth from primary inputs: fixes closer to primary inputs are preferred. Under "Err. diag. time", "1st" is the runtime for diagnosing the first bug trace, while "Total" is the runtime for diagnosing all 32 traces. The comparison between the first and total diagnosis time shows that diagnosing the first trace takes more than 30% of the total diagnosis time in all the benchmarks. The reason is that the first diagnosis can often localize errors to a small number of sites, which reduces the search space of further diagnoses significantly. Since CoRé relies on iterative diagnosis to refine the abstraction of signatures, this phenomenon ensures that CoRé is efficient after the first iteration. As Table I shows, error diagnosis is still the bottleneck of the CoRé framework. We also observe that fixing some bugs requires a large number of gates and support variables in their resynthesis netlists because the bugs are complex functional errors at the RTL.

VI. CONCLUSIONS

In this paper we propose a framework, called CoRé, to correct functional errors in digital circuits relying only on error traces. This framework exploits both satisfiability and observability don't-cares, and it uses an abstraction-refinement scheme to achieve better scalability. To support the resynthesis task required in the framework, we propose an encoding of resynthesis information, called PBDs, and use it in our innovative resynthesis techniques. Because CoRé does not rely on specific error models, it offers more error-correction capabilities than many previous solutions. The experimental results show that CoRé can produce a modified netlist which eliminates erroneous responses while maintaining correct responses. In addition, CoRé supports both combinational and sequential error repair, and it can be easily adopted in most verification flows.

REFERENCES

[1] M. F. Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith and M. Abadir, "Debugging Sequential Circuits Using Boolean Satisfiability", *ICCAD'04*, pp. 44-49.

[2] K.-H. Chang, I. L. Markov and V. Bertacco, "Fixing Design Errors with Counterexamples & Resynthesis", *ASPDAC'07*, pp. 944-949.

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

[4] R. Rudell and A. Sangiovanni-Vincentelli, "Multiple-Valued Minimization for PLA Optimization", *IEEE TCAD*, Sep. 1987, pp. 727-750.

[5] S. Sinha, "SPFDs: A New Approach to Flexibility in Logic Synthesis," *Ph.D. Thesis*, University of California, Berkeley, May 2002.

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

[7] S. Staber, B. Jobstmann, R. Bloem, "Finding and Fixing Faults", *CHARME'05*, Springer-Verlag LNCS 3725, pp. 35-49.

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

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

[10] Y.-S. Yang, S. Sinha, A. Veneris and R. E. Brayton, "Automating Logic Rectification by Approximate SPFDs", *ASPDAC'07*, pp. 402-407.

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

[12] MVSIS, `http://embedded.eecs.berkeley.edu/Respep/Research/mvsis/`

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