Counterexample-Guided Diagnosis

Heinz Riener*
*Institute of Space Systems,
German Aerospace Center, Germany
heinz.riener@dlr.de

Goerschwin Fey*†
†Faculty of Mathematics and Computer Science,
University of Bremen, Germany
goerschwin.fey@dlr.de

***Abstract*—In this paper, we propose a counterexample-guided diagnosis approach to identify faults in circuit designs described as net-lists on the gate-level. Given a faulty net-list and a logic specification of the correct, intended behavior of the circuit, the diagnosis algorithm iteratively computes the exact set of fault candidates, i.e., a subset of the circuit's gates at which all counterexamples can be rectified. The algorithm equips SAT-based diagnosis with systematic counterexample generation. In each iteration, an over-approximation of the fault candidates is computed and a new counterexample is generated such that at least one of the fault candidates can be excluded in the next iteration. The algorithm terminates if no such counterexample exists and no remaining fault candidate can be excluded. The number of counterexample generated is not minimal and, thus, we additionally provide a counterexample reduction algorithm to post-process the set of generated counterexamples and obtain some insight in how many counterexamples are sufficient to exactly pinpoint a fault. We evaluate counterexample-guided diagnosis for a set of benchmark circuits and provide a comparison to an exact algorithm that uses a state-of-the-art QSAT oracle. The accuracy of both algorithms is per design equal, whereas counterexample-guided diagnosis significantly outperforms the QSAT-based diagnosis algorithm.**

I. Introduction

Design debugging is a tedious and cumbersome process that requires a substantial amount of human work and time. The overall effort spent in debugging is often reported to account for half of the total development time and a quarter of the total development budget. Formal approaches to diagnose faults based on *Boolean satisfiability* (SAT) [1] and *quantified SAT* (QSAT) have been proposed. SAT-based algorithms over-approximate the set of fault candidates considering a fixed set of counterexamples; however, finding the right counterexamples to obtain a tight over-approximation is complicated and typically not addressed. Alternatively, QSAT-based algorithms compute the exact set of fault candidates with respect to all possible input assignments but do not scale well with the size of the circuit.

In this paper, we propose a counterexample-guided diagnosis algorithm using a SAT oracle that computes the exact set of fault candidates, but scales significantly better than QSAT-based algorithms. Given a faulty circuit design described as a net-list on the gate-level and a logic specification of the correct, intended behavior, e.g., obtained from a golden reference circuit or a test suite, the proposed algorithm uses SAT-based diagnosis to compute an over-approximation of the fault candidates from an initial set of counterexamples. The over-approximation is then iteratively improved by systematically constructing new counterexamples, adding them to the initial set, and recomputing the set of fault candidates. Each counterexample is constructed in such a way that at least one fault candidate is removed in the next iteration. The algorithm terminates if no such counterexamples can be generated anymore and thus the exact set of fault candidates has been determined.

The advantages of the counterexample-guided diagnosis algorithm can be summarized as follows: 1.) the algorithm has anytime character, i.e., a first, coarse over-approximation of the fault candidates is obtained fast and stepwisely improved when time progresses. This is in contrast to QSAT-based algorithms that compute the exact solution in one long-running computation. Our algorithm can be interrupted by a user at anytime, e.g., when the user has time constraints for debugging the circuit, and an over-approximation of the exact set of fault candidates is obtained. 2.) If a good initial set of counterexamples is known and provided for the first approximation of the fault candidates, the algorithm performance can be significantly improved similarly to SAT-based algorithms. 3.) Counterexample-guided diagnosis computes the exact set of fault candidates and significantly outperforms QSAT-based algorithms.

We provide a prototype implementation of the proposed counterexample-guided diagnosis algorithm that takes as input an *and-inverter graph* (AIG) and computes fault candidates in terms of a list of potentially faulty gates of the AIG as output. We evaluate the run-time of our prototype implementation for a set of selected combinational circuits and compare the performance to an exact algorithm that uses a state-of-the-art QSAT oracle.

The remainder of the paper is structured as follows: in Section II and Section III, we present related work and preliminaries, respectively. In Section IV, we give a brief introduction to circuit verification and diagnosis including a motivating example. Section V is dedicated to the counterexample-guided diagnosis algorithm, whereas Section VI presents experimental results. Section VII concludes the paper.

II. Related Work

Approaches to debugging are manifold. Early attempts to fault localization stem from diagnostic reasoning developed in the field of artificial intelligence. These approaches are inspired by ideas and notions from philosophy. *Diagnosis from first principles* rests on the observation that when a change applied to a system results in a system that no longer exhibits the specification violation, the changed components can be used as a characterization of the fault. From this perspective, the changed components are one potential cause for the observed specification violation. Reiter [2] showed that finding changes of minimal size, called minimal diagnosis, is NP-complete and proposed a hitting set algorithm to enumerate all minimal diagnoses. The general theoretical framework of diagnosis from first principles is flexible — a component may be a gate of a digital circuit or a sub-circuit — and only requires that the system has to be described in a suitable logic. Smith et al. [1] developed a diagnosis procedure for circuits that uses a SAT oracle. Today, techniques following similar ideas are typically categorized as *SAT-based diagnosis* techniques. In this context, many heuristics [3], [4], [5] to improve the accuracy of diagnosis have been proposed. In this paper, we develop a counterexample-guided diagnosis approach based on a SAT oracle. Our algorithm, in contrast to standard SAT-based diagnosis, is exact and has anytime character.

Most similar to our approach, Sülflow and Fey [5] generate counterexamples targeted to refute diagnosis. Instead of applying a QSAT oracle, however, our implementation "unrolls" the quantified Boolean formulæ and instantiates the all-quantified variables with concrete values to obtain a SAT instance. Moreover, all interaction with the SAT oracle are done via API which enables incremental SAT solving.

Other applications of QSAT to fault diagnosis for circuits include compacting the time frame for sequential circuits [6] and extracting corrections [7] for circuits.

III. Preliminaries

A. SAT and QSAT

Given a Boolean formula f over Boolean variables $v_0, \dots, v_{n-1}$, the SAT problem asks whether an assignment $a = a_0 \cdots a_{n-1}$ to the variables exists under which f evaluates to true, i.e., $f(a) = 1$. If such an assignment exists, f is *satisfiable* and otherwise *unsatisfiable*.

The QSAT problem generalizes SAT. Given a quantified Boolean formula f, where some variables $b_0, \dots, b_{k-1}$ are existentially ($\exists$) or universally ($\forall$) *bounded* and some variables $v_0, \dots, v_{n-1}$ occur *free*, the QSAT problem asks whether an assignment $a = a_0 \cdots a_{n-1}$ to the free variables exists such that f evaluates to true. The SAT problem can be understood as a special case of QSAT, where all variables are implicitly existentially quantified. SAT is the canonical complete problem for NP, whereas QSAT is the canonical complete problem for PSPACE. Thus decision procedures for SAT and QSAT unleash the power to express and decide all instances of problems in NP and PSPACE, respectively.

Despite the problem's complexities, effective decision procedures for SAT and QSAT, called *oracles*, have been developed for many years. SAT oracles operate on Boolean formulæ in *conjunctive normal form* (CNF), i.e., a conjunction of disjunctions of literals, where a *literal* is a variable or its negation. QSAT oracles operate on *prenex CNF* (PCNF) of form $Q_0 b_0 \cdots Q_{k-1} b_{k-1} . M(v_0, \dots, v_{n-1}, b_0, \dots, b_{k-1})$ with quantifiers $Q_i \in \{\exists, \forall\}$ and bounded variables b_i, $0 \leq i \leq k-1$, and free variables v_i, $0 \leq i \leq n-1$. The prefix $Q_0 b_0 \cdots Q_{k-1} b_{k-1}$ is called the *prenex* and $M(v_0, \dots, v_{n-1}, b_0, \dots, b_{k-1})$ is called the *matrix*.

B. Circuits and Specifications

A *circuit* is a net-list on the gate-level. For the sake of simplicity, we only consider combinational circuits in this paper; however, all presented algorithms can be generalized to sequential circuits. A combinational circuit is a directed acyclic graph with nodes corresponding to logic gates and edges corresponding to wires connecting them. The sources of the graph are the circuit's *primary inputs* (PIs) and the sinks are the *primary outputs* (POs).

Each circuit can be encoded as a Boolean formula in CNF over Boolean variables, e.g., using the *Tseytin transformation* [8]. More succinct encoding, e.g., [9], have been proposed. Let C be a circuit, we write $C(x, y)$ to denote a Boolean formula C in CNF over lists x and y of Boolean variables that correspond to the circuit's PIs and POs, respectively. In the following, we will not distinguish between the circuit C and the circuit's CNF encoding $C(x, y)$ when it is clear from the context.

A logic specification is a Boolean formula that similarly to a CNF encoding of a circuit describes an input-output relation between PIs and POs. A specification may be the CNF encoding of a golden, reference implementation of a circuit or a logic encoding of a test suite that incompletely specifies the correct outputs only for some inputs. The accuracy of automated diagnosis strongly depends on the completeness of the specification. Faults that are not "detected" by the specification cannot be identified with diagnosis. In the following, we do not deal with finding a logic specification of a circuit, but assume that a logic specification that allows to identify the faults of interest is always available.

IV. Verification and Diagnosis

A. Formal Verification

Let C be a faulty circuit and R be a logic specification that describes the correct, intended behavior of C. *Formal verification* attempts to guarantee that the circuit adheres to its logic specification, i.e., the POs of C and R have to be equal when executed on the same PIs. This can be formalized as

$$\forall x. \exists y. (C(x, y) \wedge R(x, y)) \tag{1}$$

and checked using a QSAT oracle.

In practice, instead of formally verifying the correctness of a circuit, often *model checking* is applied to check whether a counterexample to correctness exists, i.e.,

$$\exists x, y, y'. (C(x, y) \wedge R(x, y') \wedge y \neq y'). \tag{2}$$

Eq 2 avoids the costly quantifier alternation of Eq 1 and allows to implement a decision procedure utilizing a SAT oracle. Additionally, to deciding satisfiability, SAT oracles typically also generate assignments to the variables as a

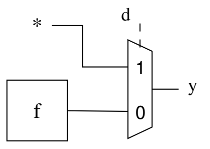

Fig. 1. Injection of Boolean values using abnormal variables.

certificate for the satisfiability of the formula. In terms of Eq 2, an assignment (x, y'), i.e., a pair of an input assignment that exhibits a specification violation and the corresponding correct values at the outputs, serves as a *counterexample*. A counterexample is then used as a starting point for manually debugging the circuit.

B. Fault Diagnosis

In order to reduce the effort of manually debugging circuits, automated diagnosis algorithms have been developed. Automated diagnosis, e.g., [1], attempts to highlight the "interesting" gates of a circuit, i.e., those gates that should be examined in debugging, for a designer by computing the subset of gates that allow for rectifying the circuit's input-output relation when changed. However, diagnosis does not provide a fix for the circuit.

Let C be a faulty circuit and R be a logic specification that describes the correct, intended behavior of C. Diagnosis algorithms deal with the identification of a subset of the circuit's components to be diagnosed, called *diagnoses*, that when changed correct the circuit's input-output relation such that Eq 1 is satisfied. We consider gates as components and the terms *diagnosis* and *fault candidate* are understood as synonym. For each potentially faulty gate in the circuit to be diagnosed, an additional abnormal variable is introduced. The abnormal variables are used to inject non-deterministic values for the corresponding gates when assigned to true. This can be best understood as instrumenting the output of each gate with a multiplexer controlled by the abnormal variable as shown in Fig 1. Suppose that f is a gate (or a component) of the circuit. When $d = 0$ the gate (or component) behaves normally, i.e., y is assigned to the output of f; however, when $d = 1$ then y is assigned to a non-deterministic Boolean value. During reasoning the non-determinism is resolved when needed, i.e., y takes the "right" value to satisfy the constraints imposed by Eq 1 if possible. In practice, the multiplexer construction shown in Fig 1 is not implemented into the circuit, but added to the CNF when encoding the circuit.

The diagnosis problem can then be described as

$$\exists d.\forall x.\exists y.(C(x, y, d) \wedge R(x, y)), \tag{3}$$

where d represents the list of abnormal variables introduced into the CNF encoding for diagnosis. An assignment to d is a diagnosis for the faulty circuit, i.e., the set of gates with $d = 1$ is a fault candidate. Note that multiple faults are allowed such that several gates together form a fault candidate. Following Occam's razor, simpler diagnoses are always preferred and thus often an additional cardinality constraint $|d| \leq l$ is added to Eq 3 to ensure that at most l variables are assigned to 1. To obtain all possible diagnoses, Eq 3 is iteratively checked, where each assignment obtained for d is blocked until no more assignments exist and the formula becomes unsatisfiable. This can be implemented utilizing an *incremental* QSAT oracle, where additional constraints can be added after each check for satisfiability.

As in verification, the costly quantifier alternation of Eq 3 is undesirable. Thus, SAT-based approximations for Eq 3 exist. Let C be a faulty circuit and suppose that a fixed-size list of $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ of counterexamples for C is known, e.g., obtained by model checking or simulation of the design. The diagnosis problem of Eq 3 can be approximated as

$$\exists d.(\bigwedge_{i=0}^{n-1} C(x_i, y_i, d)) \tag{4}$$

and checked with a SAT oracle. The CNF encoding of the circuit is cloned n-times and the PIs and POs are constrained to the counterexamples, respectively. Note that the abnormal variables d are shared over all clones to diagnose the same gates considering all counterexamples but the non-deterministic Boolean values injected at the diagnosed gates may change for each counterexample. Again, to obtain all diagnoses, respectively, all fault candidates, Eq 4 has to be checked multiple times, where each time the extracted assignment to d has to be blocked, until the formula becomes unsatisfiable. Since Eq 4 considers only a fixed number of counterexamples in contrast to Eq 3, an over-approximation of the fault candidates is computed.

C. Motivating Example

In this section, we illustrate SAT-based diagnosis for a simple example, a faulty realization of the circuit `c17` from the ISCAS'85 benchmark suite. The circuit has 5 PIs, 2 POs, and consists of 6 NAND gates. The correct, golden reference circuit is shown in Fig 2 (on the top) together with the faulty realization to be diagnosed (on the bottom). The faulty realization is annotated with a counterexample obtained from model checking the circuit. The counterexample shows the inputs and outputs of all gates (for the faulty realization in red and the reference circuit in blue).

In order to over-approximate the fault candidates of the faulty circuit with respect to the given counterexample, the schema described in the previous section can be equipped: the PIs and POs are assigned to the values of the counterexample. All gates are considered potentially faulty and thus instrumented with abnormal variables, conceptually shown in Fig 3. As in Eq 4, an assignment to the abnormal variables can be computed utilizing a SAT oracle.

The SAT oracle assigns $d_2 = 1$ and computes a single fault candidate for g_2 when invoked. No other diagnoses are obtained. This corresponds to the exact location of the fault. Note that, due to the given counterexample, only the gates g_2, g_4, and g_5 are interesting for diagnosis; all other gates of the reference circuit and the faulty realization produce the expected outputs. Moreover, since both POs are affected by the

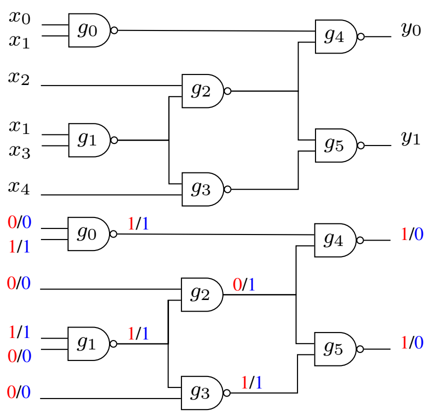

Fig. 2. The correct, golden reference circuit c17 (top) and a faulty realization annotated with a counterexample (bottom).

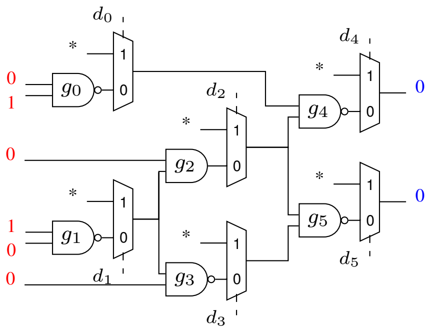

Fig. 3. The faulty realization of Fig 2 instrumented for diagnosis.

fault, no diagnosis can be generated for g_4 and g_5, respectively, and thus g_2 is the only fault candidate obtained. In this special case only one counterexample is enough to exactly pinpoint the fault. For more realistic examples, often more than one counterexample is needed.

V. Counterexample-Guided Diagnosis

In this section, the counterexample-guided diagnosis algorithm is described that combines SAT-based diagnosis with a systematic approach to generate counterexamples. The algorithm iteratively improves an over-approximation of the fault candidates until the exact set is obtained. The overall diagnosis algorithm is described in Section V-A, whereas counterexample generation and a post-processing algorithm to reduce the number of counterexamples, called counterexample reduction, are presented in Section V-B and Section V-C, respectively.

A. Overview of the Diagnosis Procedure

Algorithm D describes the overall approach to counterexample-guided diagnosis. The generation of counterexamples is separately described as Algorithm G invoked by Algorithm D.

Algorithm D (*Counterexample-Guided Diagnosis*). Given a faulty net-list C and a logic specification R, this algorithm determines the exact set of fault candidates Δ for C with respect to R.

D1.[Initialize.] Set $\Delta := \emptyset$ and $T := \emptyset$.

D2.[ModelCheck.] Encode C and R into a Boolean formula (as in Eq 2) and apply a SAT oracle to compute an initial counterexample that is added to T.

D3.[Diagnose.] Set $F := \emptyset$. Encode C for each counterexample in T with abnormal variables d as a Boolean formula (as in Eq 4). Apply a SAT oracle to decide satisfiability of the Boolean formula. If satisfiable, extract the assignment to d from the oracle's certificate and add the set $\{i \mid d_i = 1\}$ of gate ids diagnosed to be faulty to F. Lastly, block the assignment to d in the SAT oracle and goto D3. Otherwise, if unsatisfiable, goto D4.

D4.[Refute.] If $F \backslash \Delta = \emptyset$, goto D5 . For each $f \in (F \backslash \Delta)$, try to compute a counterexample by applying Algorithm G to C, R, and f. If Algorithm G returns a new counterexample, add it to T, and goto D3. Otherwise, if $\perp$ is returned, add f to Δ and proceed with the next diagnosis.

D5.[Terminate.] Return Δ. ▮

Note that initially a single counterexample is generated by model checking the faulty circuit with respect to its logic specification. Alternatively, if a set of counterexamples is available, they can also be used as a starting point for diagnosis.

B. Counterexample Generation

To *refute* a potential diagnosis d, that corresponds to a set of gates in the circuit, a counterexample (x, y) is required that reveals that the faulty circuit C and the logic specification R produce different outputs regardless which values are injected at the outputs of the gates in d. This can be formalized as

$$\exists x.\forall u.\exists y, y'.(C(x, y', u) \wedge R(x, y) \wedge y \neq y'), \tag{5}$$

where u denotes a list of fresh variables that corresponds to the outputs of the gates in d. Each variable of u corresponds to one specific gate in d.

Due to the quantifier alternation in Eq 5, a QSAT solver is needed for refuting a diagnosis. However, the number of gates in a diagnosis is typically low such that the all-quantifier can be removed by reproducing the matrix of the quantified formula $2^{|d|}$-times while replacing u with each of the possible evaluations. The resulting Boolean formula can be checked with a SAT oracle.

For instance, assume that d contains a single gate, then Eq 5 is semantically equivalent to

$$\begin{aligned} \exists x, y, y', y''.(y \neq y' \wedge y \neq y'' \wedge \\ C(x, y', 0) \wedge C(x, y'', 1) \wedge R(x, y)). \end{aligned} \tag{6}$$

Algorithm G describes an approach to generate a counterexample to refute a certain diagnosis utilizing a SAT oracle.

Algorithm G (*Refute Diagnosis*). Given a faulty net-list C, a logic specification R, and a diagnosis d, this algorithm determines a counterexample c that refutes d when used in SAT-based diagnosis or returns $\perp$ if no such counterexample exists.

D1.[Encode.] Encode C and R into a Boolean formula and introduce for the gates in d a list u of fresh Boolean variable that can be used to inject Boolean values at the output of those gates (as in Eq 5). Unroll the Boolean formula by instantiating u for all possible values (as in Eq 6) such that the resulting Boolean formula needs no all-quantification of u.

D2. [Check SAT?] Apply a SAT oracle to check the satisfiability of the formula. If satisfiable extract and return the counterexample (x, y) from the oracle's certificate. If unsatisfiable, return $\perp$. ▮

C. Counterexample Reduction

Algorithm D tries to refute each diagnosis and systematically reduces the number of fault candidates. However, the number of counterexamples generated is not minimal. Newly generated counterexamples may cover previously refuted diagnoses. In an attempt to reduce the number of counterexamples as a post-processing step, Algorithm S iterates through the list of counterexample, removes one counterexample, and tests whether the set of diagnosis shrinks.

Algorithm S (*Reduce Counterexamples*). Given a faulty netlist C, a list T of counterexamples, and the corresponding set Δ of diagnoses produced with SAT-based diagnosis, this algorithm attempts to determine a subset $T' \subset T$ that generates the same set Δ of diagnosis.

S1. [Initialize.] Set $T' := T$.

S2. [Test.] For each t in T, apply SAT-based diagnosis to C and $T' \backslash \{t\}$ to compute diagnosis Δ'. If $\Delta = \Delta'$, set $T' := T' \backslash \{t\}$.

S3. [Terminate.] Return T'. ▮

Algorithms S invokes SAT-based diagnosis at most $|T|$-times. The reduced list T' of counterexamples produced is a local minimum, but depending on the order the algorithm examines the counterexamples, and consequently does not guarantee that the global minimum is computed. Alternatively, to obtain the exact solution a minimal cover can be computed.

VI. Experimental Results

The described exact counterexample-guided diagnosis approach was implemented and evaluated for a selected subset of combinational circuits given as *and-inverter graphs* (AIGs). The benchmarks were taken from the ISCAS'85 and EPFL benchmark suites. The ISCAS'85 benchmarks were first translated from the BENCH format in functionally equivalent AIGs utilizing ABC; the EPFL benchmarks are already provided as AIGs. The number of PIs, POs, gates, and levels reported in the following always refers to the translated AIG circuits. For each circuit, 100 faulty versions were generated with random fault injection. As fault model, we consider for all wires stuck-at-0 and stuck-at-1 faults as well as missing or additional logic negations. Each faulty version contains exactly one fault.

All experiments were conducted on a quad-core Intel® Core$^{\text{TM}}$ i5-2520M CPU with 2.50GHz and 8GB RAM. As SAT oracle, MiniSAT 2.2.0 was used. Since we concentrate on single faults, cardinality constraints that restrict the search to diagnoses of size 1 have been added.

A. Comparison to QSAT-based Diagnosis

To analyze the correctness of the approach, counterexample-guided diagnosis was compared to an exact diagnosis algorithm that implements the approach described in Eq 3 using a state-of-the-art QSAT oracle. Different QSAT oracles have been evaluated, however, DepQBF 5.0 [10] performed best in our experiments. Nonetheless, the QSAT oracle timed out for all considered circuit designs except for `c17`. The timeout was set to 10 minutes per benchmark. The quality of the diagnoses of the exact algorithm and counterexample-guided diagnosis are equal, i.e., for all faulty versions, both algorithms compute the same diagnoses. We compared and verified the results for `c17`.

In an attempt to find a suitable time bound for the circuit `c432`, we computed the exact set of fault candidates for one of the faulty version of `c432` without time restrictions in 7 hours using DepQBF. Other QSAT oracles did not terminate within the same time.

B. Evaluation for Combinational Benchmark Circuits

We have used a set of benchmarks to evaluate the run-time of the counterexample-guided diagnosis approach. Table I lists details for those benchmarks. The table is built as follows: the first column names the benchmark followed by four columns that describe the benchmark's characteristics, that are, the number of primary inputs (#PIs), the number of primary outputs (#POs), the number of gates (#Ands), and the number of levels (#Lvl) of the circuit. The next four columns list the number of generated counterexamples and the number of diagnosis, respectively, on average (μ) and the standard derivation (σ). The last two columns list the run-times required for model checking (MC), i.e., for generating the first counterexample, and for diagnosis (D), i.e., generating the fault candidates and refuting them.

In general, only a few counterexamples were sufficient to exactly pinpoint the set of fault candidates, i.e., for all benchmarks less than 10 counterexamples were generated. The run-times for model checking and diagnosis were for all circuit designs presented in the table acceptable. The counterexample-guided diagnosis approach, however, timeouts for more complex designs, e.g., the 16-bit multiplier circuit `c6288` from the ISCAS benchmark suite could not be analyzed. Although model checking and SAT-based diagnosis (with respect to multiple counterexamples) succeeds on the design in a few seconds, generating counterexamples to refute a diagnosis is too complex.

The number of counterexamples generated with counterexample-guided diagnosis is not minimal and, thus, we apply the counterexample reduction algorithm, Algorithm S, described in Section V-C as a post-processing step. Table II gives an overview of the results for the same benchmark set. The table is built similar to Table I. The characteristics of the benchmarks are identical to the previous table and thus have been omitted. The first column names

TABLE I
DIAGNOSIS FOR SELECTED BENCHMARKS

Name	**#PIs**	**#POs**	**#Ands**	**#Lvl**	**#Cex**		**#FC**		**Time**	
					μ	σ	μ	σ	MC	D
`int2float`	11	7	260	16	1.72	0.85	9.38	6.78	0.00	0.07
`priority`	128	8	978	250	4.96	2.28	7.44	10.16	0.02	1.08
`dec`	8	256	304	3	1.26	0.44	1.27	0.45	0.01	0.01
`cavlc`	10	11	693	16	1.84	0.85	10.54	7.30	0.01	0.38
`adder`	256	129	1020	255	1.68	0.68	1.90	0.89	0.02	0.39
`bar`	135	128	3336	12	2.03	0.50	3.65	2.02	0.26	91.08
`c17`	5	2	6	3	1.43	0.50	1.50	0.58	0.00	0.00
`c432`	36	7	209	42	2.22	1.07	7.10	6.18	0.00	0.10
`c499`	41	32	400	20	3.36	2.22	5.88	8.76	0.01	0.76
`c880`	60	26	327	24	2.84	1.45	5.12	3.72	0.00	0.81
`c1355`	41	32	504	26	3.72	2.35	5.95	5.95	0.01	0.91
`c1908`	33	25	414	32	2.42	1.45	6.27	8.82	0.01	0.81
`c2670`	233	140	717	21	3.84	2.53	12.41	17.48	0.03	2.54
`c3540`	50	22	1038	41	2.22	1.23	8.26	7.03	0.10	12.65
`c5315`	178	123	1773	38	2.95	2.26	6.94	5.22	0.06	7.55
`c7552`	207	108	2074	29	3.76	2.38	10.25	12.59	0.06	18.53

TABLE II
COUNTEREXAMPLE REDUCTION

Name	**#Cex**		**#Cex$_R$**		**Time**
	μ	σ	μ	σ	
`int2float`	1.72	0.85	1.44	0.54	0.00
`priority`	4.96	2.28	3.66	1.64	0.10
`dec`	1.26	0.44	1.18	0.39	0.00
`cavlc`	1.84	0.85	1.48	0.56	0.01
`adder`	1.68	0.68	1.41	0.49	0.00
`bar`	2.03	0.50	1.41	0.49	0.05
`c17`	1.43	0.50	1.29	0.46	0.00
`c432`	2.22	1.07	2.63	1.52	0.00
`c499`	3.36	2.22	2.45	1.49	0.04
`c880`	2.84	1.45	1.92	1.01	0.01
`c1355`	3.72	2.35	3.05	1.89	0.05
`c1908`	2.42	1.45	1.91	1.08	0.01
`c2670`	3.84	2.53	2.62	1.68	0.07
`c3540`	2.22	1.23	1.69	0.69	0.02
`c5315`	2.95	2.26	2.28	1.66	0.11
`c7552`	3.76	2.38	2.63	1.52	0.24

the benchmark. The next four columns list the number of counterexamples initially generated with counterexample-guided diagnosis ($\#Cex$) and after reduction ($\#Cex_R$) on average (μ) and the standard derivation (σ), respectively. The last column lists the required run-time.

On average after reduction, only two counterexamples suffice to exactly pinpoint all fault candidates. This experiments indicates that only a few counterexamples are necessary to pinpoint faults and if the fault candidates are selected in the right order, e.g., by considering the circuit's structure, several SAT calls can be avoided.

VII. Conclusion

In this paper, we proposed a counterexample-guided diagnosis approach that computes the exact set of fault candidates for combinational circuit designs described as net-lists on the gate-level. A prototype implementation of the approach using a SAT oracle was presented and evaluated for a set of selected benchmarks taken from the ISCAS'85 and EPFL benchmark suites. The experimental results indicate that the approach significantly outperforms other exact algorithms based on QSAT solving, while producing exact fault candidate sets. We assume that the approach's performance can even be improved when a suitable heuristic for generating initial counterexamples is applied.

VIII. Acknowledgments

This work was supported in part by the European Union (Horizon 2020 IMMORTAL project, grant no. 644905).

References

[1] A. Smith, A. G. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 10, pp. 1606–1621, 2005.

[2] R. Reiter, "A theory of diagnosis from first principles," *Artificial Intelligence*, vol. 32, no. 1, pp. 57–95, 1987.

[3] G. Fey and R. Drechsler, "Finding good counter-examples to aid design verification," in *ACM & IEEE International Conference on Formal Methods and Models for Co-Design*, 2003, p. 51.

[4] A. Sülflow, G. Fey, C. Braunstein, U. Kühne, and R. Drechsler, "Increasing the accuracy of SAT-based debugging," in *Design, Automation and Test in Europe*, 2009, pp. 1326–1331.

[5] A. Sülflow, G. Fey, and R. Drechsler, "Using QBF to increase accuracy of SAT-based debugging," in *International Symposium on Circuits and Systems*, 2010, pp. 641–644.

[6] H. Mangassarian, A. Veneris, and M. Benedetti, "Fault diagnosis using quantified Boolean formulas," in *IEEE Silicon Debug and Diagnosis Workshop*, 2007.

[7] S. Staber and R. Bloem, "Fault localization and correction with QBF," in *International Conference on Theory and Applications of Satisfiability Testing*, 2007, pp. 355–368.

[8] G. S. Tseytin, "On the complexity of proofs in propositional logics," in *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*. Spinger, 1983, vol. 2, originally published in 1970.

[9] N. Eén, A. Mishchenko, and N. Srensson, "Applying logic synthesis for speeding up SAT," in *International Conference on Theory and Applications of Satisfiability Testing*, 2007, pp. 272–286.

[10] F. Lonsing, F. Bacchus, A. Biere, U. Egly, and M. Seidl, "Enhancing search-based QBF solving by dynamic blocked clause elimination," in *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, 2015, pp. 418–433.