Increasing the Accuracy of SAT-based Debugging

André Sülflow*, Görschwin Fey*, Cécile Braunstein†, Ulrich Kühne* and Rolf Drechsler*
*Institute of Computer Science
University of Bremen, 28359 Bremen, Germany
Email: {suelflow,fey,ulrichk,drechsle}@informatik.uni-bremen.de
†Laboratoire LIP6-SoC
University Paris VI, 75252 Paris, France
Email: cecile.braunstein@lip6.fr

***Abstract*—Equivalence checking and property checking are powerful techniques to detect error traces. Debugging these traces is a time consuming design task where automation provides help. In particular, debugging based on *Boolean Satisfiability* (SAT) has been shown to be quite efficient. Given some error traces, the algorithm returns fault candidates. But using random error traces cannot ensure that a fault candidate is sufficient to explain all erroneous behaviors.**

Our approach provides a more accurate diagnosis by iterating the generation of counterexamples and debugging. This increases the accuracy of the debugging result and yields more valuable counterexamples. As a consequence less time consuming manual iterations between verification and debugging are required – thus the debugging productivity increases.

I. Introduction

The increasing complexity of today's designs requires automation throughout the whole design process. Verification is one of the major bottlenecks. There exist effective techniques as constrained random simulation or formal verification to show faulty behavior, but analyzing the reason is often still a manual and time consuming debugging task. Techniques to automate debugging have been proposed. Among these, debugging based on *Boolean Satisfiability* (SAT) [1] has been shown to be quite robust and applicable to a variety of design scenarios from diagnosis to debugging properties. An overview is provided in [1], [2]. Given a set of failure traces or counterexamples, the method computes the set of fault candidates that can explain an observed error.

In principle, the more counterexamples are used, the more accurate is the debugging result, i.e. the set of fault candidates is reduced. Moreover, when all counterexamples are used, each fault candidate is *complete:* modifying a fault candidate is sufficient to correct all errors. But obviously, using all counterexamples is not feasible in practice. Instead, a subset has to be chosen, where choosing the best subset is NP-complete [3]. Therefore the counterexamples are typically chosen randomly and the fault candidates are not guaranteed to be *complete*.

The technique of [4] uses such randomly generated counterexamples for debugging, applies re-synthesis to provide a corrected circuit and iterates the verification process. This loop continues until all erroneous traces are fixed. But choosing bad counterexamples may cause many iterations and modifications in many different parts of the circuit. Both is not desirable. Therefore, strengthening the debugging result can lead to less re-synthesis steps and to less changes to the design.

Without an automated re-synthesis step the loop between verification and repair requires the time-consuming intervention of a designer. In this case better counterexamples are even more desirable, raising the debugging productivity and decreasing the design time.

In this paper, we present a technique to ensure that each fault candidate is sufficient to fix all counterexamples, i.e. *is complete*. Moreover, the process terminates when no more qualitatively different counterexamples are available. Hence, we often need less counterexamples than in a random approach. The technique can be applied in domains like combinational or sequential equivalence checking and property checking.

Our method combines counterexample generation and debugging in a single algorithm. First, a faulty circuit is checked against a specification and a counterexample is extracted. Then, SAT-based debugging provides a fault candidate that is sufficient to fix the counterexample. Finally, additional counterexamples that cannot be fixed using the same fault candidate are determined and the process iterates.

The proposed algorithm uses a Boolean encoding of three-valued logic for checking completeness of fault candidates. Thus some erroneous behavior may be left undetected due to the restricted power of three-valued simulation. Re-synthesis [4] or a symbolic method is required for further strengthening the diagnosis. In the following a *complete* fault candidate denotes completeness with respect to three-valued simulation.

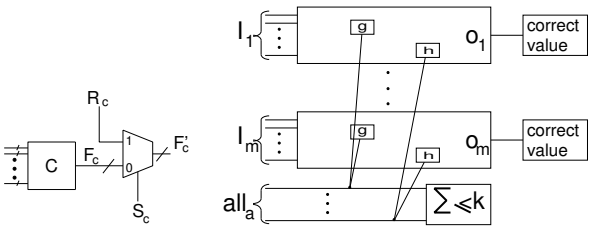
With each new counterexample the accuracy increases by excluding fault candidates that cannot fix all faulty behaviors. If no additional counterexamples are found, each fault candidate is *complete*. In an alternative application scenario our technique starts from a given set of counterexamples and finds qualitatively different counterexamples if available.

In experimental studies combinational and sequential circuits are considered. The efficiency is shown on single and multiple faults. In comparison to a set of random counterexamples the accuracy significantly increases.

In the following we present our technique for *Equivalence Checking* (EC) on the gate level, but the extension to property checking or hierarchical descriptions is straight forward.

The paper is structured as follows: Section II introduces preliminaries on three-valued logic and SAT-based debugging.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) and by Concept Engineering GmbH, Freiburg, Germany within the project Herkules under contract no. 01 M 3082.



(a) Correction (b) Problem instance

Fig. 1. Combinational debugging

Section III shows limitations of previous approaches. Section IV introduces our approach. Experimental results are summarized in Section V and Section VI concludes.

II. Preliminaries

A. Three-Valued Logic

Considering circuit *Observability Don't Cares* (ODC) in formal hardware verification leads to stronger counterexamples [5], [6] and faster verification engines [7], [8]. Each counterexample containing unknown (X) values gives a better explanation of the faulty behavior since it does not depend on the signals or components with unknown values. ODCs are often computed in a post-processing step [6] or by a modified SAT solver [5], [8]. But, these approaches do not allow to use ODCs for logical reasoning.

Here, the three-valued logic defined over $\{0, 1, X\}$ requires a direct encoding in *Conjunctive Normal Form* (CNF) for a standard SAT solver. Accordingly the modeling of gates and components in the CNF formula has to be adjusted. We encode three-valued logic similar to [7] by using two variables for each Boolean signal.

Considering X values requires additional overhead in CNF encoding. Our current implementation requires 3 to 5 times more variables and clauses for each EC instance on the gate level. The factor depends on the types of gates in the circuit, but is linear in the number of gates.

B. SAT-based Debugging

In this section the basics of SAT-based debugging [1] are reviewed. The algorithm accepts a faulty circuit, a set of input stimuli (counterexamples) and the expected correct output responses as input. This information is used to compute gates as fault candidates that may fix the circuit.

In general, a circuit is divided into components. Depending on which elements are chosen as components, the granularity of the debugging result differs. Typical choices are gates or expressions, but also hierarchical or structural information are taken into account [9], [10], [1], [2]. In our examples we consider gates to keep the presentation simple.

For each component extra *correction logic* is inserted. The original output function F_c of component C is replaced by F'_c as shown in Figure 1(a). The select line S_c of the additional multiplexer controls F'_c such that if S_c is activated, then $F'_c = R_c$, otherwise $F'_c = F_c$. R_c is an unconstrained variable that can have any value assigned. S_c is also called *abnormal predicate*. When activated, S_c leads to a non-deterministic output function of component c. Therefore, F'_c may behave non-deterministically.

Given a set of m counterexamples, for each counterexample a combinational debugging instance is created as shown in Figure 1(b): the faulty circuit with additional correction logic is constrained to the inputs (I) of the counterexample and the correct output responses (O), given from e.g. a golden specification. The same abnormal predicate is used for each instance of a single component. Therefore, activating an abnormal predicate leads to a non-deterministic behavior of a component in all instances. The instance is unsatisfiable if all abnormal predicates are deactivated, since no correction can be performed.

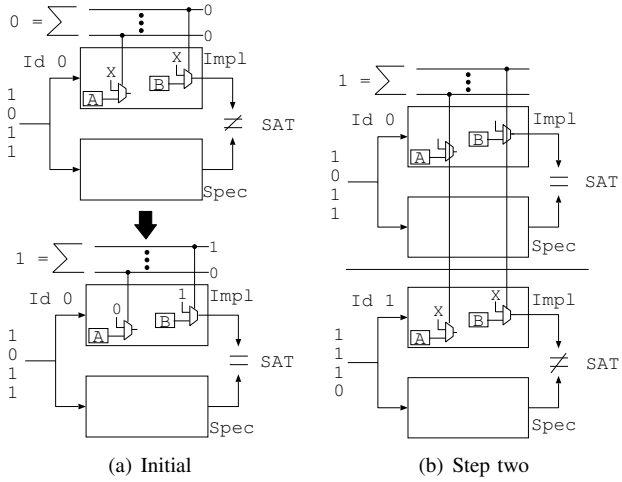
During debugging the underlying SAT solver searches for satisfying assignments by activating some of the correction logic. The algorithm increases the number of activated abnormal predicates from 1 to k. Here k is the minimal cardinality leading to a satisfiable instance. Then all fault candidates under the limit k may be extracted by adding a blocking clause for each fault candidate obtained. A fault candidate is a single component or a set of components in case of multiple faults ($k > 1$). The constrained model is translated into *Conjunctive Normal Form* (CNF) and given to a SAT solver to check if the instance is satisfiable (SAT) or unsatisfiable (UNSAT).

The extension to sequential circuits is straightforward: the circuit is unrolled and the same abnormal predicates are used in all time steps and for all replicated instances [11]. For property debugging [2] instead of applying a set of correct output responses, the property is applied as reference that has to be fulfilled. To consider more complex components instead of gates all outputs of a single component are connected to the same abnormal predicate [9], [2]. Additionally, techniques to improve the efficiency have been reported.

III. Limitations

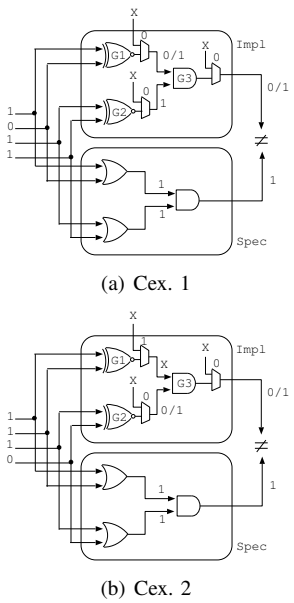
SAT-based debugging has one major drawback: it is *exact*, but not *complete*. That is, for a given set of m counterexamples each fault candidate is sufficient to fix these observations – each fault candidate is *exact*. But it is unknown whether a fault candidate is sufficient to fix any faulty behavior wrt. the specification, i.e. whether a fault candidate is *complete*. Therefore, a designer has to apply SAT-based debugging in a loop: (1) create a counterexample, (2) diagnose fault candidates, (3) fix the design and start again by checking if the specification is now fulfilled. Therefore in each loop the design has to be fixed until the specification is completely fulfilled. The task may be automated [4], but the automation may lead to "unwanted" designs. That is, due to the iterative application of re-synthesis, changes to many different parts of the design may happen. So strengthening the debugging results – by reducing the number of fault candidates – before fixing the design can lead to more accurate fixes.

The approach in [3] heuristically chooses fault candidates that are fixes to a set of counterexamples. But, no guarantee is given that one of the counterexamples contains more information than another one. So, it may happen, that m counterexamples are heuristically chosen that lead to almost the same



(a) Initial (b) Step two (c) Final

Fig. 2. Overview



(a) Cex. 1

(b) Cex. 2

Fig. 3. Example

fault candidates. When using too many counterexamples, the size of the SAT instance increases, without reducing the fault candidates.

Regarding completeness previous approaches cannot decide whether the set of computed fault candidates for a given set of counterexamples may fix all faults in the design. So there may be still an uncovered scenario that is not fixable using the final set of fault candidates. Thus, a designer starts debugging with this insufficient set of counterexamples and fault candidates, fixes the observed errors and, after this time consuming process, finds that other qualitatively different errors remain. The process iterates.

IV. Increasing Accuracy

In this section an algorithm is proposed, that resolves the above limitations of SAT-based debugging by (1) finding qualitatively different counterexamples and (2) returning only complete fault candidates. The proposed algorithm uses X values to identify redundant counterexamples and to prevent them from further occurrence.

Starting from a faulty implementation and a specification the debugging algorithm computes all fault candidates that can fix *all* faulty behaviors. The specification may be given as a golden specification for equivalence checking or a property for *Bounded Model Checking* (BMC) [12].

Each fault candidate in the final set may be used for resynthesis to compute a concrete repair [4].

In the following the algorithm is explained for combinational EC on the gate level. Adding the extensions mentioned in Section II-B to our approach is straightforward. In Section IV-A the basic idea is presented and explained by an example in Section IV-B. The complete algorithm is proposed in Section IV-C. Finally, a discussion of possible drawbacks and limitations is given in Section IV-D.

A. Idea

Usually counterexample generation and debugging algorithm are separated. That is, first a number of counterexamples are generated and afterwards the debugging algorithm computes the fault candidates. That offers to use different techniques in counterexample generation (e.g. testing or simulation) and for SAT-based debugging. On the other hand, due to the separation it is not possible to control the generation to get a set of "useful" counterexamples. Therefore the idea is to combine counterexample generation and debugging in a single algorithm.

In the following we focus on *equivalence checking* (EC) wrt. a golden specification. The extension to property checking along the lines of [2] is straightforward.

B. Completeness Check

Figure 2 shows an overview of the process. First, an EC instance is created as shown in Figure 2(a), top. A faulty circuit with correction logic and a specification without correction logic are created and a comparator is connected to the primary outputs. Additionally, a limitation constraint is applied and forces exactly 0 abnormal predicates to behave abnormal – the behavior without additional correction logic. Therefore, any satisfiable solution is a counterexample.

In the figure, the instance is satisfiable, a counterexample is extracted and debugged afterwards (Figure 2(a), bottom). The counterexample is applied and the comparator at the primary outputs is constrained to force the behavior of the specification. This instance becomes satisfiable by increasing the number of activated abnormal predicates to 1. Thus, the counterexample is fixable by activating exactly one abnormal predicate. An explicit enumeration of all solutions is not necessary here, it is sufficient to compute the minimal cardinality $k = 1$.

Now, in parallel to the first instance ($Id\,0$), a second EC instance ($Id\,1$) is added (Figure 2(b)). The same abnormal predicate is given to a single component in both instances and the primary inputs R_c on the correction logic are forced to X. An X denotes all possible values and thus, all erroneous behaviors that can be fixed at this place. Hence, it excludes paths that are already fixable with some value for X. If

an X value propagates to one of the primary outputs the instance is unsatisfied, because the miter output is constrained to be a specified (non-X) value 1 or 0. Therefore no new counterexamples are found.

A SAT solver checks whether activating one of the components in $Id\ 0$ still leaves a counterexample in $Id\ 1$ that cannot be fixed. Thus, a satisfying solution proves a previous fault candidate not *complete*.

The second counterexample is extracted and $Id\ 1$ is constrained for debugging. In the figure, the limitation of $k = 1$ active abnormal predicates is assumed to be sufficient. Each satisfying solution denotes a fault candidate for both counterexamples. In the general case, incrementing k may be necessary (as discussed in Section IV-D).

A third instance $Id\ 2$ is added as shown in Figure 2(c). The SAT solver proves the model to be unsatisfiable. That is, there are no more counterexamples that are not already fixable under the constraints of $Id\ 0$ and $Id\ 1$.

Finally, all fault candidates can be enumerated by running an all solution SAT solver on the model of Figure 2(b). The activation of any fault candidate in the final set allows to fix all counterexamples.

Example 1. *Consider the computed counterexample of $Id\ 0$ in Figure 3(a). SAT-based debugging returns $G1$ and $G3$ as fault candidates under the limitation $k = 1$. Due to the limitation to 1, either the abnormal predicate of $G1$ or $G3$ is activated and therefore one of the components is allowed to behave non-deterministically. Both components behave correct, if a 1 is introduced as correction value R_c.*

After fixing the first counterexample a second instance $Id\ 1$ is added. Here, for $G1$ and $G3$ an X is assumed on the primary input R_c of the correction logic. Activating $G3$ leads to an unsatisfiable instance, but activating $G1$ leads to another, unfixed counterexample as shown in Figure 3(b).

The second counterexample may be fixed at $G2$ or $G3$, but $G2$ does not fix the first counterexample. Therefore, fault candidate $G3$ is the only one that fixes both counterexamples.

A third instance is added, but no more counterexamples exist after considering $G3$ as fault candidate. The analysis is complete and $G3$ is a fault candidate that fixes all faulty behaviors.

C. Algorithm

In the following the complete algorithm to obtain only fault candidates that can explain all error traces is presented. The input for the algorithm shown in Figure 4 is a faulty circuit ($\mathcal{F}$) and the specification to be fulfilled ($\mathcal{S}$) (Line 1). The algorithm proceeds in the following main steps:

1) Initialization
2) Compute limitation k and counterexamples
   a) Add new EC instance
   b) Find next counterexample. If there are no more counterexamples, return all fault candidates.
   c) Extract counterexample
   d) Compute k and proceed with 2.(a)

```
1  function debugging (F,S)
2  Id = 0;
3  k = 0;
4  do {
5   // Add Model
6   (C,l,ABs) = addECInstance(F,S,Id);
7
8   ∀a ∈ ABs: primaryInput(a) = X;
9   insert limitation(|ABs| = k);
10  insert forceFaultyBehavior(C);
11
12  if (solve() == UNSAT)
13   return extractAllFaultCandidates();
14
15  // Extract Counterexample
16  cex = extractCounterExample(Id,l);
17  remove ∀a ∈ ABs: primaryInput(a) = X;
18  remove forceFaultyBehavior(C);
19
20  // Diagnosis
21  constrainCounterExample(F,cex,Id);
22  forceCorrectBehavior(C);
23  do {
24    if (solve() == SAT) break;
25    remove limitation(|ABs| = k);
26    k = k + 1;
27    insert limitation(|ABs| = k);
28  } while (k ≤ |ABs|);
29
30   Id = Id + 1;
31 } while (true);
32 end function;
```

Fig. 4. Algorithm

Step 1. Initialization – First, the counter for the number of instances (Id) and the limitation constraint are initialized (k) (Lines 2-3).

Step 2.(a) Add new EC instance – Now, a new EC instance from $\mathcal{F}$ and $\mathcal{S}$ is added and constrained as $Id\ 0$ (Line 6). The function returns the variable C that encodes the output of the comparator, the number of time frames (l) and the set of abnormal predicates (ABs). The variable C may be constrained to force faulty behavior for counterexample generation or to ensure correctness wrt. the specification.

Step 2.(b) Find next counterexample – The model is constrained to find a new counterexample in Lines 8–10. First, the additional primary inputs R_c are constrained to X for all ABs, followed by forcing exactly k components to behave abnormal. Forcing R_c on all ABs to X is not necessary, but an explicit enumeration of the current set of fault candidates is time consuming and therefore avoided here. Next, the comparator is constrained.

The instance is translated by encoding the three-valued logic into CNF and passed to a SAT solver (Line 12). If the SAT solver returns UNSAT, then there does not exist any further counterexample and all solutions can be obtained (Line 13).

Step 2.(c) Extract counterexample – The counterexample (cex) is extracted (Line 16).

Step 2.(d) Compute k – Now, the constraints on the primary inputs of the correction logic and the comparator are removed (Line 17-18).

Debugging starts by constraining cex on the model and forcing the correct behavior on the primary outputs (Lines 21-22). As long as the current limitation to k is not sufficient to satisfy the instance, k is incremented (Lines 23-28).

Finally, the instance counter Id is incremented (Line 30) and the loop continues as long as there are more counterexamples observable (Line 12).

D. Discussion

Incrementally generating the counterexamples is only one possible scenario that involves multiple calls to a costly equivalence checking procedure. Alternatively the approach decides whether a given set of counterexamples (e.g. randomly chosen) is already complete. Otherwise the algorithm generates additional counterexamples, or simply more random counterexamples are added until completeness is reached.

The proposed approach provides complete fault candidates in case of single faults as well as multiple faults. In case of multiple faults, new counterexamples may require to increase the value of k. That is, one or more counterexamples are correctable by e.g. one component only, but the new counterexample needs two or more components to be fixed (consider e.g. two unconnected parts of a single circuit). Still, the accuracy increases.

Whenever a new counterexample is created, this means that at least one of the current fault candidates cannot be a complete fault candidate. Therefore, as the number of counterexamples increases, the number of fault candidates decreases if k remains constant. Increasing k may also lead to more fault candidates in case of multiple faults.

The fault candidates in the final set are guaranteed to be complete; their number depends on the counterexamples considered.

To consider debugging wrt. a property only *addECInstance* in the algorithm has to be modified according to [2] (discussing completeness of fault candidates when considering partial specifications is beyond the scope of this work). Moreover, the model works for hierarchical debugging [9]. If one of the outputs belonging to a component is an X value, an X value is introduced on all output signals.

Due to the encoding, the primary inputs I may be X in a counterexample. Allowing X values in the primary inputs leads to more general counterexamples which is beneficial for debugging [5].

V. Experimental Results

The proposed method was evaluated on combinational and sequential circuits of the LGsynth93 and ITC-99 benchmark suite. The circuits were modified by randomly changing a set of gates. For example an AND was replaced with an OR. Gates are considered as components. In the sequential case the circuits were unrolled for ten time frames.

The experiments are conducted on a Dual-Core AMD Opteron 2220 SE (4.4 GHz, 32 GB main memory). *MiniSAT* [13] was used as underlying SAT solver.

The focus is (1) on the accuracy (Section V-A) and (2) the efficiency (Section V-B).

A. Accuracy

First, we focus on accuracy by comparing reduction of the number of fault candidates with an increasing number of counterexamples. We compare our method to standard debugging using random counterexamples from equivalence checking with a SAT solver. The counterexamples were generated by using a non-incremental SAT solver by successively adding blocking clauses. Note that, using a heuristic technique for counterexample generation, e.g. [3], does not guarantee to increase the accuracy.

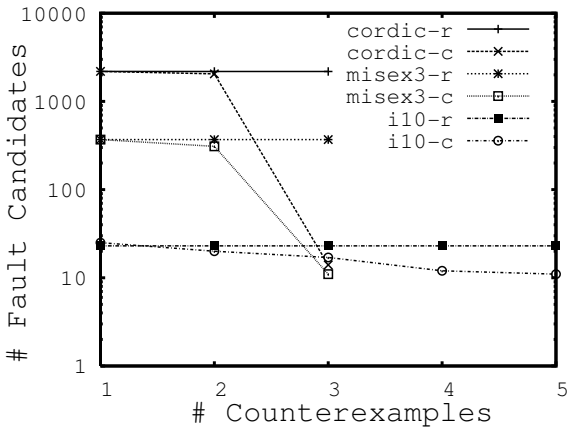


Fig. 5. Fault candidate reduction

TABLE I
Post-Processing

	Std. Diagnosis			Post-Processing		
Circuit	#C	#FC	Time (s)	#C	#FC	Time (s)
apex5	2	11	70.83	1	6	202.97
c7552	1	8	57.42	1	6	396.21
cordic	3	2184	6160.84	2	14	401.84
dalu	2	40	109.35	1	16	231.12
des	2	16	118.72	-	-	328.13
i10	7	23	161.78	8	11	782.07
misex3	2	369	6654.11	1	11	828.79
pair	2	9	24.06	1	6	64.08
seq	1	4	49.61	-	-	337.09

Figure 5 shows detailed results for a few example runs. The suffix "-*r*" marks standard debugging with random counterexamples. Suffix "-*c*" marks the proposed method.

Random counterexamples lead to similar observations for all circuits. Often additional counterexamples do not prune fault candidates. In particular for *i10*, the five random counterexamples do not lead to any reduction of fault candidates. Here, the first counterexample is sufficient to obtain the final set of fault candidates.

In contrast, the proposed method achieves a reduction of fault candidates with each new counterexample. For *cordic* and *misex* the size of the final set is less than 3% compared to the set of random counterexamples.

Table I provides further insight. Here, an alternative scenario is considered. For *Std. Diagnosis* a number of *#C* counterexamples is used that leads to *#FC* fault candidates; *Time* denotes the run time for computing all fault candidates. In a *Post-Processing* step, the new approach verifies the set of fault candidates to be complete or provides further reduction. That is, an initial debugging instance was created from the counterexamples, and the algorithm calculated additional counterexamples (second column labeled *#C*) to reduce the fault candidates (second column labeled *#FC*). Whenever additional counterexamples were generated a reduction of fault

TABLE II
SINGLE FAULTS

Circuit	#Gates	k	#C	Time (s) Debug	EC	Total	Memory (MB)
Combinational							
apex5	3940	1	2	6.28	260.69	266.97	102
c7552	4675	1	1	1.38	231.41	232.79	111
cordic	2938	1	3	16.18	159.36	175.54	123
dalu	2883	1	2	2.35	186.52	188.87	76
des	3942	1	2	5.95	7.80	13.75	80
i10	3294	1	7	36.46	151.44	187.90	176
misex3	6249	1	2	20.22	955.86	976.08	149
pair	2848	1	2	6.58	37.47	44.05	62
seq	4776	1	1	8.80	149.27	158.07	81
Sequential							
b04	821	1	1	2.35	6.88	9.23	63
b05	1198	1	1	5.79	4.77	10.56	46
b11	867	1	3	7.88	218.52	226.40	141
b12	1297	1	1	3.83	6.38	10.21	92
b14	11089	1				time out	-
b15	10513	1	1	25.76	50.12	75.89	803
gcd	1217	1	1	4.47	331.80	336.27	144
phase_decoder	1834	1	2	10.57	1275.14	1285.71	267

TABLE III
MULTIPLE FAULTS

Circuit	#Gates	k	#C	Time (s) Debug	EC	Total	Memory (MB)
Combinational							
apex5	3940	2	2	31.74	16.54	48.28	69
c7552	4675	3	18	7937.87	63918.70	71856.60	939
cordic	2938	2	6	98.99	5847.88	5946.87	862
dalu	2883	3	5	110.40	1546.92	1657.32	145
des	3942	2	1	16.48	4.12	20.60	52
i10	3294	3	4	117.27	73.55	190.82	98
misex3	6249	2	3	331.23	12058.50	12389.80	2.103
pair	2848	3	5	123.87	316.35	440.22	122
seq	4776	2	3	37.27	95.09	132.36	103
Sequential							
b04	821	3	1	22.84	4.44	27.28	63
b05	1198	2	1	4.31	7.06	11.37	52
b11	867	2	1	5.04	3.71	8.75	63
b12	1297	3	1	151.74	10.17	161.91	95
b14	11089	5	2	56687.50	15907.60	72595.10	1226
b15	10513	2	1	309.06	47.07	356.13	420
gcd	1217	2	1	13.77	4.71	18.49	82
phase_decoder	1834	3	1	48.55	8.05	56.60	137

candidates was achieved. Most impressive are the reductions from 369 to 11 fault (*misex3*) and from 2184 to 14 fault candidates (*cordic*).

Thus, randomly generated counterexamples only show some errors. Time consuming interactions between verification, debugging and correction are required to fix all errors. In contrast, our approach helps reducing the time caused by debugging. The set of counterexamples directly exhibits qualitatively different errors and only fault candidates that fix all errors are returned.

B. Efficiency

Table II and Table III show the results for single and multiple faults. The columns denote the name of the circuit, the number of gates (*#Gates*), the minimal cardinality of fault candidates (k), the number of counterexamples to reach completeness (*#C*). The time is split into time for debugging (*Debug*), equivalence checking (*EC*) and total time (*Time*). The last column denotes the required memory (*Memory*).

On combinational circuits the required number of counterexamples is moderate. In case of sequential circuits a modification in an early cycle often propagates through the circuit to fix all errors. Thus, often one counterexample is sufficient to reach completeness wrt. to the fault candidates.

The run time for debugging is typically smaller than that of EC in case of single faults. For *b14* the time out of 100,000 seconds was exceeded. When multiple faults are considered, run time and memory consumption increase significantly. In this case the time for debugging significantly exceeds the time needed for EC. Here, adapting the decision heuristic of the SAT solver as suggested in [2] can help to decrease the overall run time.

In summary, further improving the efficiency is important. But already at this stage, the number of iterations between verification and manual debugging is reduced.

VI. CONCLUSION

We proposed an approach to improve the debugging productivity by producing qualitatively valuable counterexamples. The debugging algorithm returns fault candidates sufficient to rectify all counterexamples. Thus, the fault candidates are more valuable to compute a concrete repair – less time consuming iterations are necessary.

Improving the efficiency of the algrithm is future work. Here, using dedicated tools for equivelence checking and tuning the SAT solver for the specific problem will be considered.

REFERENCES

[1] A. Smith, A. Veneris, M. Fahim Ali, and A.Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.

[2] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Trans. on CAD*, vol. 27, no. 6, pp. 1138–1149, 2008.

[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 Codesign (MEMOCODE)*, 2003, pp. 51–52.

[4] K. Chang, I. Markov, and V. Bertacco, "Fixing design errors with counterexamples and resynthesis," in *ASP Design Automation Conf.*, 2007, pp. 944–949.

[5] K. Ravi and F. Somenzi, "Minimal assignments for bounded model checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 2988, 2004, pp. 31–45.

[6] A. Groce and D. Kroening, "Making the most of BMC counterexamples," *Electronic Notes in Theoretical Computer Science*, vol. 119, no. 2, pp. 67–81, 2005.

[7] M. N. Velev, "Comparison of schemes for encoding unobservability in translation to SAT," in *ASP Design Automation Conf.*, 2005, pp. 1056–1059.

[8] S. Safarpour, A. Veneris, and R. Drechsler, "Improved SAT-based reachability analysis with observability don't cares," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 5, pp. 1–25, 2008.

[9] M. Fahim Ali, S. Safarpour, A. Veneris, M. Abadir, and R. Drechsler, "Post-verification debugging of hierarchical designs," in *Int'l Conf. on CAD*, 2005, pp. 871–876.

[10] G. Fey and R. Drechsler, "Efficient hierarchical system debugging for property checking," in *IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, 2005, pp. 41–46.

[11] M. Fahim Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith, and M.S.Abadir, "Debugging sequential circuits using Boolean satisfiability," in *Int'l Conf. on CAD*, 2004, pp. 204–209.

[12] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Design Automation Conf.*, 1999, pp. 317–320.

[13] N. Eén and N. Sörensson, "An extensible SAT solver," in *SAT 2003*, ser. LNCS, vol. 2919, 2004, pp. 502–518.