Using QBF to Increase the Accuracy of SAT-Based Debugging

André Sülflow, Görschwin Fey, and Rolf Drechsler

Institute of Computer Science
University of Bremen
28359 Bremen, Germany
{*suelflow,fey,drechsle*}*@informatik.uni-bremen.de*

Abstract. The burden of debugging significantly slows down the design process of complex systems. Only limited tool support is available and a typical experience is that fixing one problem only leads to finding the next one.
Here, we propose an approach that integrates formal verification with diagnosis. The approach is based on *Quantified Boolean Formulas* (QBF) and ensures, that counterexamples of high quality are returned. Moreover, the diagnosis algorithm only returns fault candidates that can fix all counterexamples. By this, the total number of fault candidates decreases and less iterations between verification and debugging are required.

1 Introduction

In the design process of complex systems debugging is perceived as a heavy burden. Verification methods show the existence of faults by providing traces that produce an error in terms of faulty values at the outputs. Automation for verification is available. But the following typically more time consuming task of debugging, i.e. locating and fixing the fault, remains manual work with limited tool support.

Methods to partially automate debugging have been proposed in the literature. Historically, explanation of observed errors [1–3] has been one idea while automatically locating potential fault sites, so called fault candidates, in the design has been another [4, 5].

Using formal methods, explanation [2, 3] starts with an initial counterexample and computes a set of non-counterexamples, i.e. traces that are similar to the initial counterexample but do not contradict the specification. Deriving mandatory input assignments to reproduce the faulty behavior helps the designer understanding the fault [1].

Location of fault candidates is done by diagnosis algorithms. In [5] diagnosis based on *Boolean Satisfiability* (SAT) was proposed. The approach is similar to model-based diagnosis [4] but historically evolved from the different path of diagnosing production faults in chips. SAT-based diagnosis returns fault candidates that fix all counterexamples considered using non-deterministic replacements. A specific fault model is not required. The approaches in [6, 7] use a formulation based on *Quantified Boolean Formula* (QBF) to reduce the size of the problem instance for a given set of counterexamples. Applying automatic correction [8, 9] to close the loop from verification, to diagnosis, to correction back to verification can increase the accuracy but also increases the computational costs.

Moreover, automatic corrections are not guaranteed to fix a bug in the desired way. The number of iterations until all bugs are fixed depends on the quality of counterexamples selected to perform the loop.

One major drawback of SAT-based diagnosis is the dependency of the accuracy on the quality of counterexamples. That is, SAT-based diagnosis determines a set of fault candidates with respect to the counterexamples only. Using *all* counterexamples for diagnosis is not feasible in practice. Typically, the counterexamples are chosen randomly and prior to diagnosis. Therefore, the quality of counterexamples is unknown, while the choice significantly influences diagnostic resolution. Whether all fault candidates can fix all faulty behaviors of an implementation with respect to the specification is unknown. The quality of diagnosis is affected and an over-approximation of fault candidates may be returned.

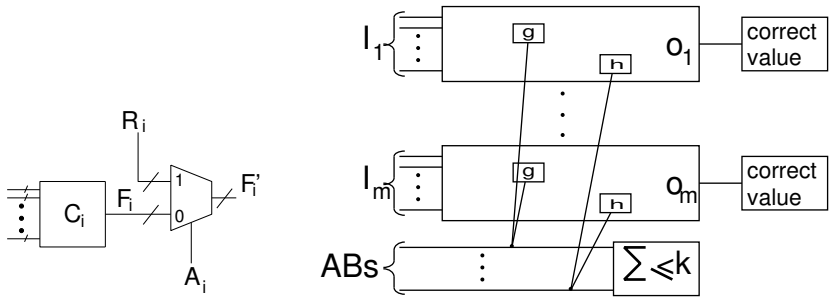
In [10] a distance metric guides the search for different counterexamples. This heuristic does not guarantee to find counterexamples that strengthen the diagnosis. *Diagnostic Test Pattern Generation* (DTPG) [11] calculates traces to distinguish given faults, but requires a specific fault model like stuck-at faults, and is therefore not model-free. For debugging the fault model is typically unknown. For higher levels of abstraction, e.g. word level, no simple fault models exist. In [12] a heuristic approach was proposed to find counterexamples. That approach is limited by the power of three-valued logic simulation, i.e. the approach is not *exact* and unfixable faulty behavior may remain undetected.

Here, we propose a framework to determine fault candidates that can fix *all* faulty behavior with respect to a functional specification provided that non-deterministic corrections are allowed. We call such fault candidates *complete* with respect to a certain length of counterexamples and to a functional specification which may be exhaustive, like in combinational equivalence checking, or partial, like in property checking. The problem is formulated in QBF to ask whether there *exists* a new counterexample where *all* potential repairs at one of the fault candidates found so far fail to heal the malfunction. The reduction of fault candidates reduces the number of time consuming iterations between fixing one bug and finding the next one. Additionally, only counterexamples covering different faulty behavior are provided to a designer.

Experimental results show a better accuracy in comparison to random counterexamples. The proposed *exact* approach is compared to the heuristic approach of [12] and proves, that the heuristic returns high quality results. For single faults the new QBF formulation is more efficient, while the heuristic is more effective for multiple faults.

We use equivalence checking at the gate level to illustrate the technique. The approach can be generalized along the lines of previous work to the sequential case [5], property checking [13], C-programs [14] and RTL debugging [6].

The paper is structured as follows: Section 2 introduces the required preliminaries to make this paper self-contained. The proposed exact approach is presented in Section 3 and evaluated in Section 4. Section 5 concludes the paper.



(a) Correction (b) Problem instance

Fig. 1. Combinational debugging

2 Preliminaries

2.1 Boolean Satisfiability

Given a Boolean expression f in *Conjunctive Normal Form* (CNF) the *Boolean Satisfiability* (SAT) problem is to decide whether there exists an assignment to the variables such that f evaluates to one. Implicitly all variables are existentially quantified. This well-known decision problem is NP-complete, but quite effective tools exist to solve SAT instances coming from formal verification, test pattern generation or similar practical problems.

A *Quantified Boolean Formula* (QBF) extends Boolean SAT by universally quantified variables. The corresponding decision problem is PSPACE-complete. Again, effective tools exist to solve QBF instances corresponding to real world problems.

2.2 SAT-based Debugging

An approach to debugging using SAT has been presented in [5]. Given an implementation of a circuit and a set of m counterexamples, i.e. input stimuli ($\{I_1, \ldots, I_m\}$) causing faulty behaviors compared to a given specification, and the expected correct output responses $\{o_1, \ldots, o_m\}$, a SAT instance for debugging is used as shown in Figure 1. For each counterexample one copy of the circuit is created, the inputs are constrained to the counterexamples and the outputs to the respective correct output responses. This is a contradiction, since the circuit produces an erroneous output in all cases. Therefore correction logic is added into the circuit for all components, e.g. g and h. A component C_i is replaced as Figure 1(a) shows. The multiplexor allows to replace the output value F_i of C by a new value R_i when the abnormal predicate a_i is asserted. The abnormal predicate is the same with respect to all counterexamples. Figure 1(b) shows the overall structure. The number of asserted abnormal predicates ABs is limited to k, i.e. k components may be changed to retrieve the correct output response.

The debugging algorithm starts with $k = 1$ and iteratively increases k until a satisfying solution is found. This yields a fault candidate FC which is a tuple of k components. Typically, not only the real fault site is returned, but several additional fault candidates. Finding the real fault within these remains to the designer.

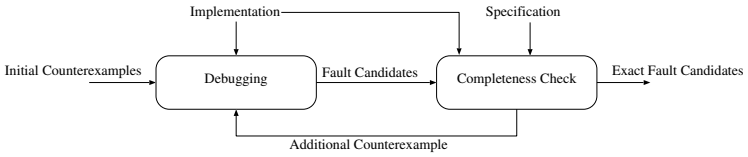


Fig. 2. Proposed debugging framework

This model-free diagnosis algorithm does not require a fault model. As a drawback fault masking may not be recognized. For example an implementation may contain two faults, but all of these faults are observed through a single output. Then, the single output is always a fault candidate of cardinality one. This is a known problem but not addressed in this work. Here we concentrate on finding high quality counterexamples.

2.3 Heuristic Approach

The approach of [12] uses three-valued logic to validate fault candidates. The validation is performed by injecting X-values at the correction logic, i.e. $R_C = X$. The X-values serve as "tokens" and mark paths that are already fixable. If an X-value is observed at an output, the approach assumes that modifying the fault candidate can create *any* value at the primary output. This overestimation may classify faulty behavior as being fixed while a more powerful reasoning engine may detect that the fix does not propagate to the outputs.

A benefit of the heuristic is that an explicit enumeration of fault candidates is not required during the completeness check. The fault candidates are implicitly enumerated by the SAT solver. Learned information is kept and may speed-up the verification for complex circuit structures significantly. Especially for multiple faults an explicit enumeration can be quite expensive, because the number of fault candidates increases exponentially with the cardinality.

Without knowing the best result, the quality of the results produced by the heuristic approach cannot be evaluated. In the following we present an *exact* approach based on QBF.

3 Exact Approach

In this section an *exact* algorithm is proposed to resolve the completeness limitation of SAT-based debugging. The algorithm ensures to compute only fault candidates that can fix *all* faulty behaviors of an implementation with respect to the specification. We call such fault candidates *complete*. Note, that a fault candidate is *complete* with respect to a given specification, to a certain length of counterexamples, and to a non-deterministic replacement. The approach combines diagnosis and formal verification in one debugging flow which places it between diagnosis (location of fault candidates) and correction (returning functionally realizable repairs).

An overview of the debugging flow is shown in Figure 2. First SAT-based debugging is applied to compute an initial set of fault candidates. Now, each fault candidate is separately checked for completeness using a QBF formulation. If one of the fault candidates is determined to be incomplete an additional counterexample is generated. The

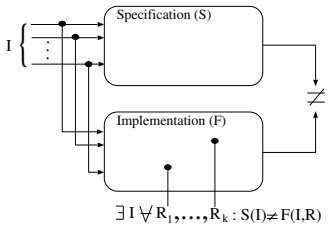


Fig. 3. QBF model

additional counterexample covers behavior that is not fixable by the fault candidate. Afterwards the new counterexample strengthens the diagnostic resolution by considering it during debugging. The new fault candidates are determined with SAT-based debugging and the completeness is checked again. The process stops if all fault candidates are verified to be complete.

Thus, the proposed debugging flow requires a formal model to verify the implementation and a specification. In this work we assume the specification given as a golden netlist for equivalence checking or a property for *Bounded Model Checking* (BMC) [15]. Equivalence checking is focus of this work, but the extension to BMC is straightforward.

In the following the details of the completeness check are presented in more detail. Section 3.1 introduces the formal model, followed by an introduction of the full algorithm in Section 3.2. The proposed approach is discussed in Section 3.3.

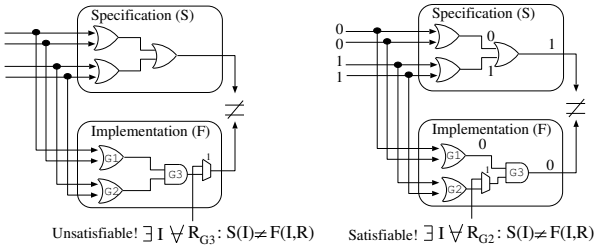
3.1 Completeness Check

Given a faulty implementation, a set of initial counterexamples, e.g. from simulation or formal verification, SAT-based debugging provides an initial set of p fault candidates of cardinality k each. That is, each fault candidate contains k components $(C_1, \cdots, C_k)$. The completeness check is applied to verify that non-deterministic behavior of a fault candidate FC fulfills the specification.

The model for completeness checking of fault candidate FC is shown in Figure 3. Given a specification $\mathcal{S}$ and an implementation $\mathcal{F}$, let I denote the primary inputs, and $R = (R_1, \ldots, R_k)$ the vector of correction values injected into the implementation at the components of the current fault candidate FC. The implementation is augmented with correction logic for the k components, the correction logic is activated. Then, the output of the specification depends on I, while the output of the implementation $\mathcal{F}$ with correction logic inserted depends on I and R. A miter circuit is created from the augmented implementation and specification. Then the QBF instance to find a new counterexample is given by

$$\exists I \forall R : \mathcal{S}(I) \neq \mathcal{F}(I, R)$$

The universal quantification of $R_1, \cdots, R_k$ determines whether none of the injected values may correct at least one counterexample. In this case FC is removed from consideration and a counterexample with uncovered faulty behavior is provided.



(a) Checking $G3$ (b) Checking $G2$

Fig. 4. Example 1

If the instance is unsatisfiable, i.e. FC is *complete* and repairs all faulty behaviors, the next fault candidate is checked. Otherwise, i.e. the instance is satisfiable, a counterexample is extracted, the verification of fault candidates stops and the additional counterexample is included in the set of counterexamples. Afterwards, all counterexamples are given to SAT-based debugging to provide an updated set of fault candidates. The process iterates.

The process stops, if the completeness of all fault candidates is proved. An *exact* set of fault candidates is provided and a designer can repair the implementation manually or automatically with e.g. [9].

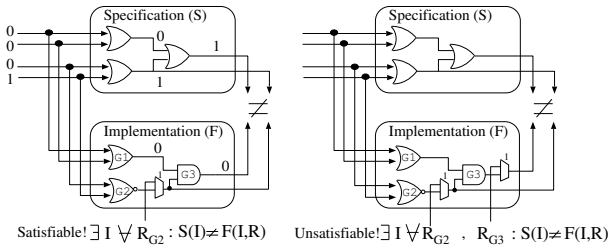
Note that no costly universal quantification of primary inputs is required, but only a few internal signals of the circuit are universally quantified. Due to the iterative approach the technique still considers all faulty behavior.

Instead of explicitly enumerating fault candidates of cardinality k, the loop to prove completeness of all fault candidates can be encoded in the QBF instance, too. But this encoding is limited to single faults and requires a constraint to restrict the cardinality of fault candidates to k and the universal quantification of *all* R_i variables corresponding to components contained in any of the fault candidates. As a result, the QBF instance becomes much more complex to solve. Therefore we do not consider this approach in the following.

Example 1. An example is shown in Figure 4, where the AND-gate $G3$ in the implementation should be an OR-gate. Debugging an initial counterexample returns the fault candidates $G2$ and $G3$ of cardinality $k = 1$. First, the completeness check is applied to $G3$ (Figure 4(a)). Because $G3$ drives the single primary output, all faulty behaviors are fixable. Therefore, the instance is unsatisfiable and $G3$ is found to be complete.

Now, the completeness of $G2$ is checked (Figure 4(b)). $G1$ drives one input of gate $G3$ in the circuit. Thus, an output value of $G1 = 0$ implies an output value of $G3 = 0$. A counterexample is determined that is not fixable for all injected values at $G2$. $G2$ can be removed from consideration, because it cannot fix *all* faulty behaviors.

But a reduction of fault candidates is not the only possible outcome. For example, the cardinality of $k = 1$ may be proved to small in case of multiple faults. That is, if no complete fault candidate of cardinality $k = 1$ exists, the algorithm provides new counterexamples as long as all faulty behaviors are covered. The completeness check ensures the determination of fault candidates of minimal cardinality.



(a) Checking $G2$ (b) Checking $G2$, $G3$

Fig. 5. Example 2

Example 2. An example for multiple faults is presented in Figure 5. In a first iteration $G2$ is determined as potential fault candidate. Now, the completeness check of $G2$ detects an uncovered scenario (Figure 5(a)). The additional counterexample is fixable by changing $G2$ and $G3$ simultaneously, only (Figure 5(b)). The cardinality is automatically increased to $k = 2$.

3.2 Algorithm

In the following the algorithm to determine an *exact* set of fault candidates is presented in more detail (see Figure 6). The input for the algorithm is a faulty circuit ($\mathcal{F}$) and the specification to be fulfilled ($\mathcal{S}$) (Line 1).

In a first step, variables are initialized: the counter for the number of counterexamples (Id), the cardinality (k) and the CNF representation of the debugging instance (cnf) (Line 2–4).

Now, an initial counterexample is created (Line 6). Any method can be used to obtain the counterexample, e.g. methods based on simulation or formal verification. In our implementation we are using equivalence checking using SAT.

While unfixable faulty behavior remains, the iteration of debugging and completeness check continues (Line 7–21).

Debugging starts with adding a debug instance for counterexample Id and all abnormal predicates (ABs) are returned (Line 8). That is, for each new counterexample the CNF for debugging is extended by a new debugging instance (see Section 2.2). Thus, the old counterexamples are kept for further diagnosis to avoid pruning of fault candidates.

In a next step, the cardinality constraint is applied and forces exactly k abnormal predicates to be one (Line 11). The instance is given to a SAT solver.

If the instance is satisfiable the completeness of fault candidates is verified (Line 12–14). If one of the fault candidates is incomplete, a counterexample is returned and strengthens the diagnosis in the next iteration (Line 13–14). If the limitation to k is not sufficient to explain the faulty behavior, i.e. the SAT instance is unsatisfiable, the current cardinality constraint is removed, k is incremented by one and debugging iterates while the maximum cardinality has not been reached. Finally Id is incremented to consider the next counterexample (Line 20).

```
1  function debugging (F,S)
2  Id = 0;
3  k = 1;
4  cnf = ∅;
5
6  cex = createInitialCounterexample (F,S);
7  while (cex != NULL) {
8   ABs = cnf.addDebugInstance(F,cex,Id);
9
10  do {
11   cnf.insertLimitation(|ABs| = k);
12   if (cnf.solve() == SAT) {
13       cex = checkFCs(cnf,ABs,F,S);
14       break;
15   }
16   cnf.removeLimitation(|ABs| = k);
17   k = k + 1;
18  } while (k ≤ |ABs|);
19
20  Id = Id + 1;
21 }
22 end function;
```

Fig. 6. Main algorithm

The completeness check using QBF is presented in Figure 7. The inputs are the CNF for debugging (cnf), a list of all abnormal predicates (ABs), the faulty implementation ($\mathcal{F}$) and the specification ($\mathcal{S}$). First, the counterexample is initialized empty (Line 2). Afterwards, it is iterated over all fault candidates and the completeness check is applied (Line 3–12). The current fault candidate (FC) of cardinality k is extracted in Line 4. Afterwards the correction logic is injected for FC, a miter circuit is created from the augmented implementation and specification and the primary inputs $R_1, \cdots, R_k$ are universally quantified (Line 6). If the QBF instance is satisfiable, i.e. FC is incomplete, a counterexample is provided and completeness check stops (Line 7–10). Otherwise, FC is blocked and the next counterexample is extracted (Line 11–12). After verifying the fault candidates all blocking clauses are removed from the debugging instance (Line 13). The algorithm returns a new counterexample or a NULL reference to show that no additional counterexample has been found (Line 14).

3.3 Discussion

The proposed completeness check is applied separately for all fault candidates. Therefore, for each fault candidate FC an equivalence checking problem in combination with universal quantification has to be solved. Thus, the QBF solver cannot re-use any learned information, due to the p independent runs. Run time limitations may occur for a large number of fault candidates but also for fault candidates of large cardinality. A QBF solver providing incremental capabilities similar to those known for SAT may alleviate this issue.

```
1  function checkFCs (cnf, ABs, F, S)
2  cex = NULL;
3  do {
4     FC = {R_i | cnf.assignment(a_i) == 1; a_i ∈ ABs}
5     //QBF check
6     qbf = createQBFInstance(∃I ∀R_1, ··· , R_k : S(I) ≠ F(I, R));
7     if (qbf.solve() == SAT) {
8        cex = qbf.extractCounterexample();
9        break;
10    }
11    cnf.addBlockingClause(FC);
12 } while (cnf.solve() == SAT);
13 cnf.removeBlockingClauses();
14 return cex;
15 end function;
```

Fig. 7. Checking completeness of fault candidates

The size of the QBF instance can be further reduced. Due to the injection and activation of the correction logic at each fault candidate, gates in the input cone of a fault candidate FC do not influence FC anymore. Therefore, structural analysis can remove gates that have no additional path to any primary output.

The QBF model does not require explicit blocking of already considered counterexamples, but blocking may be used to speed-up the verification process furthermore.

Each new counterexample increases the SAT instance for debugging, but strengthens the accuracy. Thus, the computational time for debugging is affected. To efficiently re-use the learned information of previous debug iterations, an incremental SAT solver, e.g. [16], can be used.

As already explained in Section 2.2, fault masking may occur in presence of multiple faults. The algorithm may return fault candidates of smaller cardinality than the number of faults contained in the implementation. Still, the additional counterexamples generated by the technique point to faulty behavior that remains undetected otherwise. This provides a more general view on the faulty behavior of the implementation to the designer and therefore helps during debugging.

4 Experimental Results

The proposed debugging flow was evaluated on combinational and sequential circuits of the LGsynth93 and ITC-99 benchmark suites. The faults are injected randomly by changing gates, e.g. replacing an AND gate with a NAND. Gates are considered as components. For bounded sequential equivalence checking, the circuits were unrolled for seven time frames.

All experiments are carried out on an Intel Xeon CPU (3 GHz, 32 GB main memory) running Linux. Quantor [17] (version 3.0) with PicoSAT [18] (version 632) as underlying engine was selected as QBF solver. To be comparable with Quantor, PicoSAT in non-incremental mode was used for debugging as well as for the heuristic approach. Run time was measured in CPU seconds and a time out (*T.O.*) denotes more than two hours run time.

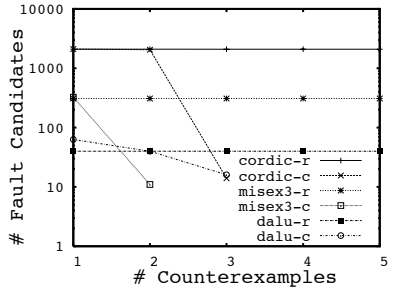


Fig. 8. Fault candidate reduction

Table 1. Post-Processing

	Std. Diagnosis			Post-Processing		
Circuit	#C	#FC	Time (s)	#C	#FC	Time (s)
apex5	2	11	153.94	1	6	67.71
c7552	2	7	129.56	2	4	66.75
cordic	2	2095	14639.00	2	14	56.56
dalu	2	40	289.13	1	16	47.36
des	2	16	497.74	1	8	187.16
i10	2	23	191.95	5	11	414.40
misex3	2	309	9185.73	1	11	308.97
pair	2	11	106.17	2	6	34.51
seq	2	4	116.61	-	-	47.63

In Section 4.1 the accuracy of the proposed approach is compared to random counterexamples. Section 4.2 focuses on the overall efficiency and compares the exact approach to the heuristic approach of [12].

4.1 Accuracy

First, randomly generated counterexamples are compared to the counterexamples of the QBF approach. The analysis is performed on combinational circuits modified by a single fault. Random counterexamples are generated using SAT-based equivalence checking.

Figure 8 shows the result. The suffixes "-r" and "-c" denote debugging with randomly generated counterexamples and the application of the exact approach, respectively[1]. Only a marginal fault candidate reduction is observable for randomly generated counterexamples. The exact approach ensures to reduce the fault candidates with each new counterexample. For *cordic* a reduction of up-to two orders of magnitude is achieved. The combination of debugging and counterexample generation always yields less fault candidates and a higher accuracy in case of single faults.

Table 1 gives further insight. Here, an initial set of two randomly generated counterexamples is applied to determine an initial set of fault candidates (Std. Diagnosis). Afterwards, the exact debugging flow is applied in a post-processing step to check the

[1] Due to the random behavior of SAT solvers different initial counterexamples are generated, e.g. for $dalu - r$ and $dalu - c$.

Table 2. Efficiency on single fault diagnosis

		Heuristic				Exact			
Circuit	#G	#C	#FC	Time	Mem.	#C	#FC	Time	Mem.
comb.									
apex5	3940	2	6	228.71	108	2	6	123.42	50
c7552	4675	1	6	508.04	125	3	**4**	69.24	49
cordic	2938	3	14	855.83	87	3	14	65.72	42
dalu	2883	3	16	365.07	104	3	16	56.15	35
des	3942	1	8	350.92	87	2	8	162.44	125
i10	3294	6	11	1415.09	263	7	11	447.87	71
misex3	6249	3	11	3314.31	611	2	11	293.83	84
pair	2848	1	8	129.12	61	2	**6**	26.92	33
seq	4776	3	4	771.66	191	2	4	95.55	68
seq.									
b04	821	1	8	365.20	122	-	-	T.O.	
b05	1198	1	2	15.56	87	1	2	5.79	113
b08	223	1	5	3.16	15	1	5	0.91	12
b10	260	2	4	6.82	23	2	4	35.14	1219
b11	867	1	6	32.77	62	1	6	35.51	1583
b12	1297	1	5	38.23	87	1	5	28.16	245
b15	10513	1	5	2213.38	759	-	-	T.O.	

completeness of fault candidates (Post-Processing). The columns in Table 1 denote the name of the benchmark (*Circuit*), the number of counterexamples (*#C*), the computed fault candidates (*#FC*) and the run time (*Time*). Column *#C* for post-processing denotes the number of additional counterexamples.

In all cases but one the number of fault candidates decreases significantly, i.e. the accuracy increases. For *seq* the randomly selected counterexamples already cover all faulty behavior. The run time for exact diagnosis was moderate. In comparison to standard diagnosis, having less fault candidates also causes smaller run times for extracting all fault candidates. In particular, extracting all 2095 fault candidates requires significantly more run time than 14 for the *cordic* benchmark.

4.2 Efficiency and Performance Comparison

The efficiency of the proposed exact approach is shown, by a comparison to the heuristic approach based on three-valued logic [12]. We consider single faults at first.

The results are presented in Table 2. The table includes information about the number of gates (*#G*), the computed total number of counterexamples (#C) and the finally determined fault candidates (*#FC*). The required run time and memory consumption in MB are presented in column *Time* and *Mem.*, respectively.

For combinational benchmarks, the QBF approach clearly outperforms the heuristic approach in all cases. Here, the encoding of the three-valued logic causes computational overhead. The accuracy, i.e. the number of finally computed fault candidates, is similar. Only for two benchmarks, i.e. *c7552* and *pair*, the exact QBF approach increased the accuracy, furthermore.

Completeness of fault candidates in sequential circuits is harder to determine. Here, the number of all universally quantified Boolean variables increases to seven, because each time frame requires another variable. Thus, the resource usage in terms of run time and memory consumption increases for QBF solving. Notable are the two time outs and the memory consumption for *b10* and *b11*.

We also performed experiments for multiple faults and hierarchical debugging [6]. In both cases the number of universally quantified variables grows significantly for the exact approach and the run time of the QBF solver increases drastically. As a result, the heuristic approach handles multiple faults and hierarchical debugging more efficiently by implicitly enumerating the fault candidates within the SAT solver.

In summary, the proposed approach based on QBF creates high quality counterexamples covering any erroneous behavior. If the exact algorithm exceeds the given resources, the heuristic of [12] still provides a very good set of counterexamples.

5 Conclusions

An *exact* approach was proposed to determine high quality counterexamples. A designer can use the high quality counterexamples to focus on different faulty behavior only. Automatically generated fault candidates are guaranteed to fix all faulty behavior up to a certain length using non-deterministic replacements. Consequently, the time consuming iterations between fixing one bug and finding the next one are reduced.

The diagnostic resolution can be further enhanced by using known correct traces. Additionally, a comparison (theoretically and empirically) to *Diagnostic Test Pattern Generation* is focus of future work.

References

1. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering **28**(2) (2002) 183–200
2. Groce, A., Kroening, D., Lerda, F.: Understanding counterexamples with explain. In Alur, R., Peled, D.A., eds.: Computer Aided Verification. Number 3114 in LNCS (July 2004) 453–456
3. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 2988 of Lecture Notes in Computer Science., Springer (2004) 168–176
4. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence **32** (1987) 57–95
5. Smith, A., Veneris, A., Ali, M.F., A.Viglas: Fault diagnosis and logic debugging using boolean satisfiability. IEEE Trans. on CAD **24**(10) (2005) 1606–1621
6. Ali, M.F., Safarpour, S., Veneris, A., Abadir, M.S., Drechsler, R.: Post-verification debugging of hierarchical designs. In: Int'l Conf. on CAD. (2005) 871–876
7. Mangassarian, H., Veneris, A., Safarpour, S., Benedetti, M., Smith, D.: A performance-driven QBF-based iterative logic array representation with applications to verification, debug and test. In: Int'l Conf. on CAD. (2007) 240–245
8. Staber, S., Bloem, R.: Fault localization and correction with QBF. In: International Conference on Theory and Applications of Satisfiability Testing. Number 4501 in LNCS (2007) 355–368

9. Chang, K.H., Markov, I., Bertacco, V.: Fixing design errors with counterexamples and resynthesis. IEEE Trans. on CAD **27**(1) (2008) 184–188
10. Fey, G., Drechsler, R.: Finding good counter-examples to aid design verification. In: ACM & IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE). (2003) 51–52
11. Zheng, F., Cheng, K.T., Yan, X., Moondanos, J., Hanna, Z.: An efficient diagnostic test pattern generation framework using boolean satisfiability. In: Asian Test Symp. (2007) 288–294
12. Sülflow, A., Fey, G., Braunstein, C., Kühne, U., Drechsler, R.: Increasing the accuracy of SAT-based debugging. In: Design, Automation and Test in Europe. (2009) 1326–1332
13. Fey, G., Staber, S., Bloem, R., Drechsler, R.: Automatic fault localization for property checking. IEEE Trans. on CAD **27**(6) (2008) 1138–1149
14. Griesmayer, A., Staber, S., Bloem, R.: Automated fault localization for c programs. Electron. Notes Theor. Comput. Sci. **174**(4) (2007) 95–111
15. Nguyen, M., Thalmaier, M., Wedler, M., Bormann, J., Stoffel, D., Kunz, W.: Unbounded protocol compliance verification using interval property checking with invariants. IEEE Trans. on CAD **27**(11) (2008) 2068–2082
16. Whittemore, J., Kim, J., Sakallah, K.: SATIRE: A new incremental satisfiability engine. In: Design Automation Conf. (2001) 542–545
17. Biere, A.: Resolve and expand. In: Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT). Volume 3542 of LNCS. (2005) 59–70
18. Biere, A.: PicoSAT essentials. In: Journal on Satisfiability, Boolean Modeling and Computation. Volume 4. (2008) 75–97