Using QBF to Increase Accuracy of SAT-based Debugging

André Sülflow, Görschwin Fey, and Rolf Drechsler
Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
Email: {suelflow,fey,drechsle}@informatik.uni-bremen.de

Abstract—Debugging significantly slows down the design process of complex systems. Only limited tool support is available and often fixing one problem 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.

(a) Correction   (b) Problem instance

Fig. 1. Combinational debugging

I. 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], [2], [3] has been one idea while automatically locating potential fault sites, so called fault candidates, in the design has been another [4], [5].

Location of fault candidates is done by diagnosis algorithms. In [5] diagnosis based on *Boolean Satisfiability* (SAT) was proposed. 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 *Quantified Boolean Formulas* (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. In [11] 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.
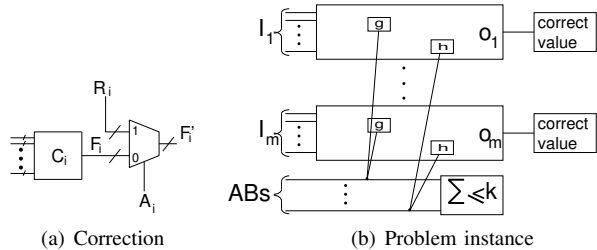
Here, we propose a framework to determine fault candidates that can fix *all* faulty behavior with respect to a functional specification provided that non-deterministic corrections are allowed. We call such fault candidates *complete* with respect to a certain length of counterexamples and to a functional specification which may be exhaustive, like in combinational equivalence checking, or partial, like in property checking. The problem is formulated in QBF to decide 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, counterexamples covering different faulty behavior are provided to a designer only.

The proposed *exact* approach is compared to the heuristic approach of [11] in the experimental section. Both approaches return high quality results. The new QBF formulation is more efficient for fault candidates of small cardinality, e.g. for single faults. However, the heuristic is more effective for hierarchical debugging and for fault candidates of large cardinality. But the exact formulation based on QBF still provides qualitatively different counterexamples.

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 [12], C-programs [13] and RTL debugging [6].

II. PRELIMINARIES

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

B. 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 erroneous output in all cases. Therefore correction logic is added 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_i by a new value R_i when the abnormal predicate A_i is asserted. The abnormal predicate for component C_i 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 among these remains to the designer.

The model-free diagnosis algorithm does not require a fault model. As a drawback fault masking may not be recognized. This is a known problem but not addressed in this work. Here we concentrate on finding high quality counterexamples.

C. Heuristic Approach

The approach of [11] uses three-valued logic to validate fault candidates. For X-values are injected at the correction logic, i.e. $R_i = X$. 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 over-estimation 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.

III. 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).

SAT-based debugging is applied to compute an initial set of fault candidates from an initial set of counterexamples. 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 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

$$\exists\, I\, \forall\, R_1, \ldots, R_k : S(I) \neq F(I, R)$$
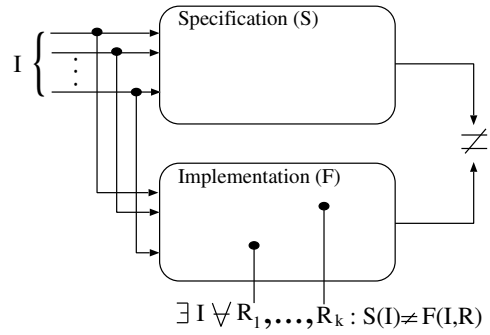
Fig. 2. QBF model

work we assume the specification given as a golden netlist for equivalence checking or a property for *Bounded Model Checking* (BMC) [14]. 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. Section III-A introduces the formal model, followed by an introduction of the full algorithm in Section III-B.

A. Completeness Check

Given a faulty implementation, a set of initial counterexamples, e.g. from simulation or formal verification, SAT-based debugging provides a set of p fault candidates of cardinality k, i.e. each fault FC candidate contains k components $\{C_1, \cdots, C_k\}$. The completeness check verifies 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 2. 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 (cf. Figure 1(a)). 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 determines whether none of the injected values $R_1, \cdots, R_k$ may correct at least one counterexample I. In this case FC is removed from consideration and a counterexample with uncovered faulty behavior is provided.

If the instance is unsatisfiable, i.e. FC is *complete* and repairs all faulty behavior, 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.

Example 1: An example is shown in Figure 3, 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
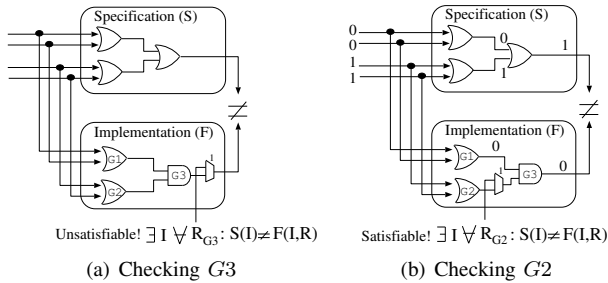
Fig. 3. Example 1

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

Unsafisfiable! $\exists I \nexists R_{G3}: S(I) \neq F(I,R)$ Satisfiable! $\exists I \forall R_{G2}: S(I) \neq F(I,R)$

of $G3$ (Figure 3(a)) is checked. 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 3(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.

Possible outcomes are a reduction of fault candidates, high quality counterexamples, and the minimal cardinality k to fix all faulty behavior. For example, the cardinality of $k = 1$ may be proved too small in case of multiple faults. That is, if no complete fault candidate of cardinality $k = 1$ exists, the algorithm provides new counterexamples until all faulty behaviors are covered. The completeness check still ensures the determination of fault candidates of minimal cardinality.

B. Algorithm

Figure 4 shows the algorithm to determine an *exact* set of fault candidates. The parameters are 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 to represent 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 II-B). 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 12–15). 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).

The completeness check using QBF is presented in Figure 5. 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

```
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. 4. Main algorithm

```
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. 5. Checking completeness of fault candidates

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).

IV. 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 replacing gates, e.g. an AND gate by a NAND. Gates are considered as components. For bounded sequential equivalence checking, the circuits were unrolled for five time frames.

All experiments are carried out on an AMD Athlon(tm) 64 X2 Dual Core processor (3 GHz, 4 GB main memory) running Linux. Quantor [15] (version 3.0) with PicoSAT [16] (version 632) as underlying engine was selected as QBF solver. ZChaff [17] with incremental SAT extension [18] was used for debugging, for the exact approach as well as for the heuristic approach. Run time was measured in CPU seconds, the memory consumption of the SAT solver in MB. *T.O.* and *M.O.* denote a time out of 24 hours and a memory out of 4 GB, respectively. The best results are marked bold.

The efficiency of both approaches is compared in Table I. Single faults are considered at first. The table shows the number of gates (#G), the computed total number of counterexamples (#C) and the finally determined fault candidates (#FC).

TABLE I
SINGLE FAULTS

Circuit	#G	Heuristic [11]				Proposed Exact			
		#C	#FC	Time	Mem.	#C	#FC	Time	Mem.
comb.									
apex5	3,938	2	6	297.33	184	3	6	**53.99**	**26**
c7552	4,674	1	6	315.60	151	4	4	**52.06**	**35**
cordic	2,938	2	14	87.08	118	3	14	**27.08**	**20**
dalu	2,883	2	16	129.99	118	3	16	**18.43**	**17**
des	3,942	3	16	188.77	204	3	8	**138.54**	**69**
i10	3,294	7	11	767.38	368	7	11	**139.80**	**52**
misex3	6,249	2	11	1,430.64	430	2	11	**126.92**	**38**
pair	2,848	1	8	37.78	60	2	6	**11.22**	**18**
seq	4,776	1	4	157.87	151	2	4	**32.45**	**37**
seq.									
b04	821	1	13	**22.91**	**66**	3	5	627.92	650
b05	1,198	1	2	7.84	95	1	2	**1.34**	**19**
b08	223	1	5	1.00	13	1	5	**0.28**	**4**
b10	260	2	5	2.27	25	3	4	**1.28**	**22**
b12	1,297	1	5	12.22	103	1	5	**3.27**	**23**
b15	10,513	1	5	2,871.78	838	1	5	**134.92**	**162**

TABLE II
MULTIPLE FAULTS

Circuit	#G	Heuristic [11]					Proposed Exact				
		#C	k	#FC	Time	Mem.	#C	k	#FC	Time	Mem.
comb.											
apex5	3,938	3	3	> 30	**3,385.10**	495	5	3	> 30	6,620.67	**270**
c7552	4,674	21	3	-	T.O.	2,122	14	3	> 30	**11,383.40**	**381**
cordic	2,938	6	2	> 30	39,241.70	755	4	2	> 30	**716.36**	**50**
dalu	2,883	5	3	> 30	7,114.22	368	6	3	> 30	**363.85**	**171**
des	3,942	2	3	9	**388.77**	237	4	3	9	1,482.48	**96**
i10	3,294	4	3	> 30	3,447.76	495	5	3	> 30	**1,724.94**	**223**
misex3	6,249	3	3	-	T.O.	727	7	3	> 30	**12,779.20**	**499**
pair	2,848	4	3	> 30	**551.89**	235	7	3	> 30	979.08	**68**
seq	4,776	6	3	-	T.O.	533	6	3	> 30	**1,599.35**	**526**
seq.											
b04	821	3	2	20	**45.35**	**152**	1	2	-	-	M.O.
b05	1,198	1	2	4	13.95	95	1	2	4	**4.69**	**35**
b08	223	2	2	5	1.81	25	2	2	5	**0.49**	**4**
b10	260	3	2	28	**7.07**	**34**	3	2	**10**	7.48	108
b12	1,297	1	**4**	> 30	**2,301.86**	**467**	1	3	-	-	M.O.
b15	10,513	5	2	9	**2,977.43**	**841**	1	2	-	-	M.O.

Required run time and memory consumption are presented in column *Time* and *Mem.*, respectively.

For combinational benchmarks, the QBF approach clearly outperforms the heuristic approach in all cases. The accuracy, i.e. the number of finally computed fault candidates, is similar. Only for three benchmarks, i.e. *c7552*, *des* and *pair*, the exact approach further increases the accuracy.

Completeness of fault candidates in sequential circuits is harder to determine. Five Boolean variables are universally quantified per component, because each time frame requires a new variable. The resource usage in terms of run time and memory consumption for QBF solving is affected. Additional counterexamples for *b04*, obtained by the proposed exact algorithm, reduce the number of fault candidates from 13 to 5. However, additional resources are required.

Experimental results on multiple fault diagnosis are presented in Table II. Due to the exponential growth of potential fault candidates, a maximum of 30 fault candidates are extracted only. For both approaches, the run time increases drastically.

For combinational circuits, the exact approach often handles the circuits very efficiently. However, not all fault candidates are proved to be complete with the exact approach. Often more than 30 fault candidates are contained and the analysis is performed partially only. Counterexamples requiring larger cardinalities may be missed.

The heuristic implicitly enumerates all fault candidates within the SAT solver. But performing a complete check with respect to all possible scenarios and all fault candidates causes overhead. For the benchmarks *c7552*, *misex3* and *seq* the time out of 24 hours has been reached.

Multiple fault diagnosis for sequential circuits shows the limitation of the exact approach. While increasing the number of universally quantified variables, the overhead significantly increases for a QBF solver. Often memory outs occurred while checking completeness of fault candidates. The heuristic still provides results within moderate run time. For example, the minimal cardinality for *b12* is determined to be 4 by the heuristic, but the exact approach computes a minimal cardinality of 3, only.

This observation has been confirmed by experiments on hierarchical designs from OpenCores [19] available in the IWLS'05 benchmark suite. Single faults are considered and the circuits are unrolled for five time frames. For the hierarchical benchmarks the activation of one component controls the activation of the correction logic on over 1000 gates. In all cases, the exact approach was not applicable and exceeds the memory limitation. The heuristic is more powerful and provides results for all benchmarks within a moderate run time.

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 [11] still provides a very good set of counterexamples. The automatically retrieved high quality counterexamples identify all aspects of faulty behavior and thereby reduce the time required for debugging.

REFERENCES

[1] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.
[2] A. Groce, D. Kroening, and F. Lerda, "Understanding counterexamples with explain," in *Computer Aided Verification*, ser. LNCS, R. Alur and D. A. Peled, Eds., no. 3114, July 2004, pp. 453–456.
[3] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 2988. Springer, 2004, pp. 168–176.
[4] R. Reiter, "A theory of diagnosis from first principles," *Artificial Intelligence*, vol. 32, pp. 57–95, 1987.
[5] A. Smith, A. Veneris, M. F. Ali, and A.Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.
[6] M. F. Ali, S. Safarpour, A. Veneris, M. S. Abadir, and R. Drechsler, "Post-verification debugging of hierarchical designs," in *Int'l Conf. on CAD*, 2005, pp. 871–876.
[7] H. Mangassarian, A. Veneris, S. Safarpour, M. Benedetti, and D. Smith, "A performance-driven QBF-based iterative logic array representation with applications to verification, debug and test," in *Int'l Conf. on CAD*, 2007, pp. 240–245.
[8] S. Staber and R. Bloem, "Fault localization and correction with QBF," in *International Conference on Theory and Applications of Satisfiability Testing*, ser. LNCS, no. 4501, 2007, pp. 355–368.
[9] K.-H. Chang, I. Markov, and V. Bertacco, "Fixing design errors with counterexamples and resynthesis," *IEEE Trans. on CAD*, vol. 27, no. 1, pp. 184–188, 2008.
[10] 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.
[11] 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. 1326–1332.
[12] 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.
[13] A. Griesmayer, S. Staber, and R. Bloem, "Automated fault localization for c programs," *Electron. Notes Theor. Comput. Sci.*, vol. 174, no. 4, pp. 95–111, 2007.
[14] M. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz, "Unbounded protocol compliance verification using interval property checking with invariants," *IEEE Trans. on CAD*, vol. 27, no. 11, pp. 2068–2082, 2008.
[15] A. Biere, "Resolve and expand," in *Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT)*, ser. LNCS, vol. 3542, 2005, pp. 59–70.
[16] ——, "PicoSAT essentials," in *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, 2008, pp. 75–97.
[17] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Design Automation Conf.*, 2001, pp. 530–535.
[18] J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A new incremental satisfiability engine," in *Design Automation Conf.*, 2001, pp. 542–545.
[19] *OpenCores*, http://www.opencores.org.