Counterexample-Guided EF Synthesis of Boolean Functions

Heinz Riener[★]
[★]DLR, e.V.
Bremen, Germany
heinz.riener@dlr.de

Rüdiger Ehlers[‡†]
[†]DFKI GmbH,
Bremen, Germany
rehlers@uni-bremen.de

Goerschwin Fey[★‡]
[‡]Institute of Computer Science,
University of Bremen, Germany
goerschwin.fey@dlr.de

Abstract. The *Exists-Forall* (EF) synthesis problem deals with finding parameters such that for all input assignments, a correctness specification is met. Many standard problems from computer-aided design and verification can be formulated as an instance of EF synthesis when a function template with holes — parameters to be synthesized — is provided. In previous work [3], we generalized the idea of EF synthesis in the context of Boolean logic by allowing existential quantification over the domain of Boolean functions (rather than Boolean variables) and presented a bounded synthesis approach guided by counterexamples to generate them using techniques from Boolean learning. We proposed to use the approach for circuit rectification and incrementally synthesized patches for digital circuits with multiple seeded faults. The approach is sensitive to a couple of parameters, including the time limits and the desired sizes of the fixes.

In this paper, we provide a detailed experimental evaluation of EF synthesis-based circuit rectification. We study the effects of modifying the parameters of the process in depth. Our results show that our circuit rectification approach even works in many cases in which large fixes are necessary, but checking the existence of small fixes where none exist is computationally costly.

1. Introduction

Satisfiability (SAT) solving has many applications in computer science and artificial intelligence. The classical SAT problem asks, given a set of constraints encoded into a logic formula over Boolean variables, whether a satisfying assignment to the variables exists such that all constraints are satisfied. SAT is the canonical NP-complete problem and was, due to its practical importance, intensively studied in the last fifty years. As a result, effective reasoning tools, called *SAT-oracles* or *SAT-solver*, are nowadays available. In practice, a SAT-oracle decides whether a Boolean formula is satisfiable and, if so, provides a satisfying assignment as witness. Beyond SAT, recently the problem of solving *Quantified Boolean Formulæ* (QBF) with exactly one quantifier alternation — the so-called *Exists-Forall* (EF) or 2QBF problem — is of major interest. Instances of 2QBF arise, e.g., in the context of parameter synthesis (or EF synthesis), where the existence of parameters is checked such that for all possible input assignments a correctness specification has to be met. Algorithms for deciding 2QBF problems using two incremental SAT-oracles were proposed, and they outperform traditional approaches for evaluating general QBF.

In the context of Boolean logic, the EF synthesis problem asks, given a Boolean predicate $\phi(\mathcal{X}, \mathcal{Y})$ over Boolean variables $\mathcal{X} = x_1, \ldots, x_n$ and $\mathcal{Y} = y_1, \ldots, y_m$, whether an assignment $\hat{\mathcal{X}} = \hat{x}_1, \ldots, \hat{x}_n$ exists such that for all possible assignments $\hat{\mathcal{Y}} = \hat{y}_1, \ldots, \hat{y}_m$, the predicate $\phi(\hat{\mathcal{X}}, \hat{\mathcal{Y}})$ holds. In many practical applications, e.g., invariant generation or repair, functions over parameters have to be found. A straight-forward approach to synthesize functions is to enumerate abstract function templates with holes — parameters to be synthesized — and apply EF synthesis to "concretize" them. Blindly enumerating function templates by size and testing them against the specification, however, requires to solve many EF synthesis problems in reasonable time. A smart enumeration strategy that avoids the enumeration of functionally equivalent expressions and learns from unsuccessful synthesis attempts is desirable.

In [3], we generalized the EF synthesis problem by allowing existential quantification over the domain of Boolean functions (rather than Boolean variables) and presented a bounded synthesis approach guided by counterexamples to generate Boolean expressions in *Disjunctive Normal Form* (DNF) using techniques from Boolean learning. In an iterative guess-and-check scheme, the approach determines a concrete Boolean function as an expression in DNF that is bounded in the number of terms or concludes that no such function exists. The approach is functional and extends ideas from parameter synthesis, where some parameters are iteratively determined in a *CounterExample-Guided Abstraction Refinement* (CEGAR) loop to guarantee that a given correctness specification is met. Instead of parameters, in our approach, a Boolean function is synthesized assuming that the function has a DNF representation. As an application, we use the CEGAR-based bounded synthesis approach to functionally rectify digital circuits with multiple seeded faults when a location at which the circuit can be rectified is known.

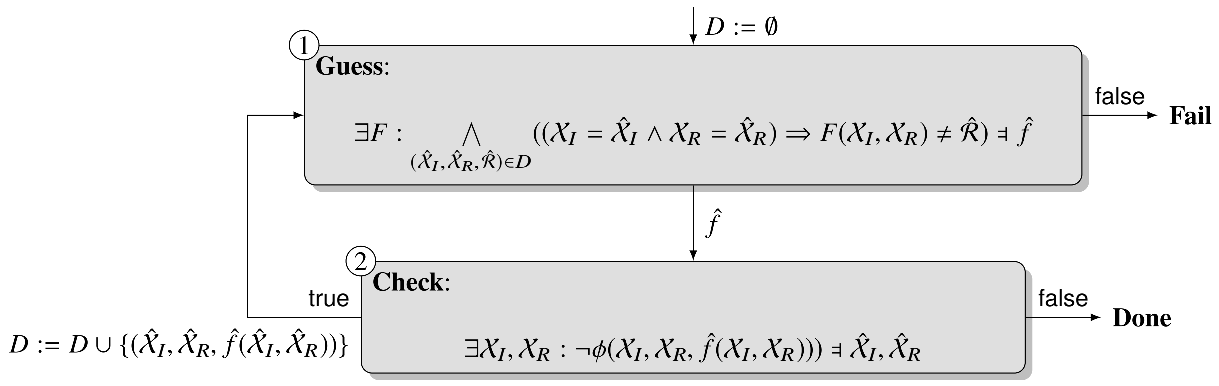
In this paper, we extend the work of [3]. We present novel experimental results and analyze the influence of relaxing the time limit and the term bound on the number of successfully rectified benchmarks. The results show that even in many cases in which large fixes are necessary, our approach finds one after short computation time. However, increasing the time limit has little influence on the percentage of the cases that can be handled by our approach.

For completeness, we repeat the description of the general logic framework for EF synthesis of Boolean functions but omit a detailed algorithmic description of the synthesis approach (which can be found in [3]).

The remainder of the paper is structured as follows: in Section 2, we describe the logic framework for synthesizing general Boolean functions (but omit the details for synthesizing expressions in DNF). In Section 3, we present circuit rectification as an application and present novel experimental results for circuit rectification in Section 4. For a detailed treatment of related work, we refer the reader to [3]. Section 5 concludes.

2. Synthesis of Boolean Functions

Let $\mathbb{B} := \{0, 1\}$. Suppose that $f : \mathbb{B}^{n+r} \to \mathbb{B}^m$ is a *multi-output Boolean function* over Boolean variables $\mathcal{X}_I := x_1, \ldots, x_n$ and $\mathcal{X}_R := x_{n+1}, \ldots, x_{n+r}$ with multiple outputs $\mathcal{Y} := y_1, \ldots, y_m$ and $\phi(\mathcal{X}_I, \mathcal{X}_R, \mathcal{Y})$ is a Boolean predicate (*a correctness specification*) which imposes logic constraints on f that have to be met by every correct realization. In the general case, the variables $\mathcal{X}_R$ are internal variables of ϕ that are totally bounded by $\mathcal{X}_I$ but may be useful to further reduce the size of a realization. In several special cases, $\mathcal{X}_R$ may be empty.



$D := \emptyset$
1
Guess:
$\exists F : \bigwedge_{(\hat{X}_I,\hat{X}_R,\hat{R})\in D} ((X_I = \hat{X}_I \wedge X_R = \hat{X}_R) \Rightarrow F(X_I,X_R) \neq \hat{R}) \dashv \hat{f}$
false
Fail
$\hat{f}$
2
Check:
true
$D := D \cup \{(\hat{X}_I, \hat{X}_R, \hat{f}(\hat{X}_I, \hat{X}_R))\}$
$\exists X_I, X_R : \neg\phi(X_I, X_R, \hat{f}(X_I, X_R)) \dashv \hat{X}_I, \hat{X}_R$
false
Done

Figure 1: CEGAR-based synthesis of Boolean functions.

The problem of Boolean function synthesis asks for finding a concrete realization f in Boolean logic such that $\phi(X_I, X_R, f(X_I, X_R))$ becomes tautological, i.e., evaluates to true for every consistent pair of concrete assignments $\hat{X}_I$ and $\hat{X}_R$. The problem can be formalized as a second-order logic query modulo Boolean logic of the form

$$\exists F : \forall X_I : \exists X_R : \phi(X_I, X_R, F(X_I, X_R)) \dashv f, \tag{1}$$

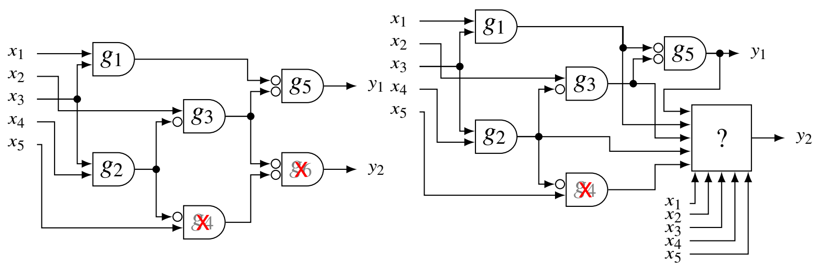
where F is a second-order variable. On success, a model f for F is a Boolean function that satisfies all constraints imposed by ϕ. Queries of such a form are beyond the capabilities of classical SAT and QBF-solvers. SAT-solvers cannot deal with quantifier alternations, whereas neither SAT nor QBF-solvers do allow to quantify over Boolean functions.

Exists-forall synthesis for Boolean functions. Suppose that the specification ϕ is provided, a CEGAR-based exists-forall synthesis approach approximates the domain of the $\forall$-quantified variables in Eq. 1 with a subset of selected sample points. The sample points are iteratively generated using the guess-and-check scheme shown in Figure 1. In each iteration, a Boolean function $\hat{f}$ is guessed that is consistent with all sample points in an initially empty database D. The function is then checked against ϕ. If the check fails, a counterexample is generated and added to D. The process terminates if either no new Boolean function $\hat{f}$ consistent with the sample points in D can be guessed, which proves that no such function exists (**Fail**), or if no counterexample exists that refutes $\hat{f}$, which proves that the guessed function is a correct solution (**Done**).

3. Circuit Rectification

In this section, we describe a circuit rectification approach as an EF synthesis problem to incrementally synthesize patches for faulty digital circuits. Such an approach may be applicable in a logic synthesis design flow, e.g., to correctly implement last-minute engineering change orders or to rectify faulty circuits due to errors in electronic design automation tools.

Suppose that C is a faulty combinational circuit with correctness specification S, e.g., provided as a reference circuit or a set of logic constraints. Moreover, suppose that also a fixed set of locations $l_1, \ldots, l_m$ in C is known at which the circuit can be rectified. The locations may be manually chosen by a designer or automatically computed using techniques from automated fault localization. We introduce a Boolean predicate $\mathsf{Rectify}(X_I, X_R, f(X_I, X_R))$ that evaluates to true if C produces the



i	x_1	x_2	x_3	x_4	x_5	g_1	g_2	g_3	g_4	g_5	g_6	**?**
1.	0	0	0	0	0	0	0	0	1	0	**1**	true
2.	1	1	0	0	1	0	0	1	0	0	**0**	$(\overline{x}_1)$
3.	1	1	1	1	1	1	1	0	0	1	**1**	$(\overline{g}_3)$
4.	0	0	0	1	1	0	0	0	1	0	**0**	$(\overline{x}_4 \cdot \overline{g}_3) \vee (g_2)$
5.	0	0	0	1	0	0	0	0	1	0	**1**	$(\overline{x}_5 \cdot \overline{g}_3) \vee (g_2)$

Figure 2: Incremental synthesis of a circuit patch by CEGAR-based EF synthesis.

same outputs as S on input x when circuitry described by the Boolean function $f : \mathbb{B}^{n+r} \longrightarrow \mathbb{B}^m$ is plugged into the circuit as a replacement for the outputs of the gates at the locations $l_1, \ldots, l_m$. In this settings, the inputs of f could be any subset of the primary inputs and the circuit gates not in the output cone of gate l. Selecting the primary inputs only suffices to guarantee completeness of the approach with respect to the term bound. The more signals are additionally selected, the better the odds to find a *small* rectification by re-using existing logic. The functional synthesis problem can be formalized as a second-order logic query and is an instance of the generalized EF synthesis problem introduced in the previous section. The Boolean predicate Rectify encodes the circuit C and ensures that the Boolean variables $\mathcal{X}_R$ can only take values compatible with the circuit structure and the input assignments to $\mathcal{X}_I$. Once the $\mathcal{X}_I$ values are chosen, the values for $\mathcal{X}_R$ are totally determined such that the innermost existential quantifier in Eq. 1 can be neglected and the logic formula has exactly one quantifier alternation.

Example 1 *As an example, consider the combinational circuit shown in Figure 2, e.g., obtained after optimization. The circuit has two faults marked for the reader with* x *symbols (in red) at the respective gates. The fault locations, however, are actually not known to the designer. A specification in terms of an unoptimized reference circuit is available and can be used for functional verification and debugging, but does not shed light on the exact problem for the behavioral mismatch of the two circuits, e.g., because they are structurally too different. An automated fault localization tool reveals that the circuit can be fixed at the output of gate* g_6*. The designer is now left with the problem of determining a fix that rectifies the input-output behavior of the circuit at* g_6*. The problem of finding a possible replacement can be formalized as an instance of Eq. 1, as shown in Figure 2, where a new combinational block ? has to be synthesized using primary inputs or internal signals. The table on the right shows the progress of the bounded synthesis algorithm for synthesizing a patch at location* g_6*, where the counterexample that disproved functional equivalence of the optimized and unoptimized circuits is provided to the algorithm as an initial sample point. Each line shows one mined input-output sample from the truth table of the Boolean function to be synthesized and the corresponding learned circuit patch in DNF (with at most* 3 *terms) that is plugged into ? in the next iteration. After 5 iterations, a correct patch is found and the algorithm terminates.*

Table 1: Number of successfully rectified benchmarks for different restrictions on the number of terms to be synthesized (timeout = 10s).

Name	**#Terms**														
	1	2	3	4	5	6	7	8	9	10	**20**	30	40	50	100
c17	0	9	9	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**
c432	1	6	**7**	6	**7**	**7**	**7**	**7**	6	6	5	4	1	1	1
c1355	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0
c7552	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0
cavlc	0	0	2	6	8	8	5	6	5	8	**10**	**10**	**10**	**10**	9
ctrl	6	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**
int2float	0	0	6	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	6
priority	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0
router	0	1	2	2	2	2	**3**	**3**	**3**	**3**	**3**	**3**	2	2	2
Total	8	28	38	45	48	48	46	47	45	48	**49**	48	44	43	38

4. Experimental Evaluation

Benchmarks. No standard benchmark set for circuit rectification (with known faults) is available. Thus, we use the ISCAS'85 benchmarks as well as the EPFL combinational benchmark suite[1] with seeded random faults. The rectification approach does not rely on any assumptions about the type of the seeded faults. However, for the experiments, we seeded stuck-at faults as well as single negations on multiple locations. To keep our implementation simple and flexible, all circuits were first translated to *And-Inverter Graphs* (AIGs) utilizing ABC [1]. From each circuit considered, 10 faulty versions are generated. Each contains at least three faults at random locations; however, a single location for rectifying the circuit is known. These locations were computed using the automated fault localization approach presented in [4]. Benchmarks which cannot be rectified at a single location were not considered. For instance, in case of the multiplier `c6288`, multiple seeded faults always propagate to multiple outputs on independent paths such that a single rectification is not possible. Since multiple faults are seeded at random locations which are potentially remote from each other, in several cases large logic cones have to be synthesized in order to correctly rectify the circuits.

Experiments. All experiments are conducted on a quad-core Intel® Core™ i5-2520M CPU with 2.50GHz and 8GB RAM running Arch Linux kernel 4.5.4-1. The check of a candidate solution is implemented leveraging the combinational equivalence checker of ABC via API. ABC produces counterexamples only in term of assignments to the primary inputs and primary outputs. Thus each counterexample is re-simulated on the circuit graph to obtain values for the outputs of all internal gates. For incrementally learning the candidate functions, we used the SAT-solver MiniSAT 2.2.0[2] via its API. The runtime required for combinational equivalence checking and re-simulation can be neglected for all considered benchmarks; runtime is mainly dominated by Boolean learning.

Fast rectification with different term restrictions. Table 1 lists the number of successfully rectified benchmarks for different term restrictions with a fixed timeout of 10 seconds per faulty circuit. The first column names the benchmark, whereas each of the following columns is dedicated to a certain term restriction. The last row lists the total number of successfully rectified faulty circuit. A figure in bold face indicates a maximum in the respective row. If no faulty circuit of a benchmark

[1]The EPFL Combinational Benchmark Suite, `http://lsi.epfl.ch/benchmarks`

[2]MiniSAT [2], `http://minisat.se/MiniSat.html`

Table 2: Number of successfully rectified benchmarks considering different timeouts and a fixed number of terms (5 terms on the left and 20 terms on the right).

Name	**Timeout**				
	10	30	60	120	**180**
c17	**10**	**10**	**10**	**10**	**10**
c432	**7**	**7**	**7**	**7**	**7**
c1355	0	0	**1**	**1**	**1**
c2670	0	0	0	0	**1**
c3540	0	0	**1**	**1**	**1**
c7552	**1**	**1**	**1**	**1**	**1**
cavlc	8	**10**	**10**	**10**	**10**
ctrl	**10**	**10**	**10**	**10**	**10**
int2float	**10**	**10**	**10**	**10**	**10**
priority	0	1	**2**	**2**	**2**
router	2	**3**	**3**	**3**	**3**
Total	48	52	55	55	**56**

Name	**Timeout**				
	10	30	60	**120**	**180**
c17	**10**	**10**	**10**	**10**	**10**
c432	5	**6**	**6**	**6**	**6**
c1355	0	0	0	0	0
c2670	0	0	0	0	0
c3540	0	0	0	0	0
c7552	**1**	**1**	**1**	**1**	**1**
cavlc	**10**	**10**	**10**	**10**	**10**
ctrl	**10**	**10**	**10**	**10**	**10**
int2float	**10**	**10**	**10**	**10**	**10**
priority	0	0	**1**	**1**	**1**
router	3	4	4	**6**	**6**
Total	49	51	52	**54**	**54**

could be rectified, the corresponding benchmark was omitted from the table. This happened in case of six benchmarks; namely, `c499`, `c880`, `c1908`, `c2670`, `c3540`, and `c5315`.

Clearly, using only one or two terms is not enough to rectify many circuits. The number of terms, however, should not be chosen too high, otherwise the performance of the Boolean learning decreases as expected, since the SAT instances become too large to be solved in time. The peak of the number of successfully rectified benchmarks is achieved with a term restriction of 20 terms, but an almost identical result is already achieved when a restriction to five terms is considered. In case of `c1335`, `c7552`, and `priority` only one faulty circuit could be successfully rectified before the time limited exceeded. The experiment shows that 49 out of 170 faulty circuits, i.e., 28.8% of all faulty circuits, could be successfully rectified in up to 10 seconds.

Five-term and twenty-term rectification with increasing timeouts. Table 2 lists the number of successfully rectified benchmarks with a term restriction of five terms and twenty terms, respectively, for different timeouts. The table is built as follows: the first column names the benchmarks, whereas the other columns list the number of successfully rectified benchmarks for the timeouts 10, 30, 60, 120, and 180 seconds. Increasing the time bound has a similar effect as relaxing the term restrictions — more faulty circuits can be successfully rectified. However, if all successfully rectified benchmarks are considered from the previous experiment (which are 53 out of 170), when the term restrictions are not considered, increasing the time does not improve the result much. Only eight faulty circuits were additionally rectified. Moreover, the function of the number of successfully rectified benchmarks with time as the dependent variable saturates quickly. From this, we conclude that the remaining faulty circuits cannot be rectified with five terms and the SAT-solver times out on the computationally hard problem of proving that the faulty circuits are not rectifiable.

5. Conclusion

We present a novel experimental evaluation of EF synthesis of Boolean function using Boolean learning techniques in the context of circuit rectification. In a case-study with the ISCAS-85 and the EPFL benchmarks, the influence of the term bound as well as the time limitation was analyzed. The experimental evaluation demonstrates two important strengths of the synthesis approach: (1.) About a quarter (28%) of all benchmark circuits can be rectified quickly (in less than 10 seconds), which was already conjectured in [3]. (2.) The synthesis approach can generate large-scale patches for circuit rectification. This result opposes our initial expectations of the approach.

We scaled our rectification approach to up to 100 terms, and still were able to successfully produce a substantial number of patches in reasonable time.

Acknowledgment

This work was supported by the European Union (grant no. 644905) and the Institutional Strategy of the University of Bremen, funded by the German Excellence Initiative.

References

[1] Brayton, Robert K. and Alan Mishchenko: *ABC: an academic industrial-strength verification tool*. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 24–40, 2010.

[2] Eén, Niklas and Niklas Sörensson: *An extensible SAT-solver*. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 502–518, 2003.

[3] Riener, Heinz, Rüdiger Ehlers, and Goerschwin Fey: *CEGAR-based EF synthesis of Boolean functions with an application to circuit rectification*. In *Asia and South Pacific Design Automation Conference*, 2017. To Appear.

[4] Riener, Heinz and Görschwin Fey: *Exact diagnosis using Boolean satisfiability*. In *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016*, pages 53–58, 2016.