An Automated Testing and Debugging Toolkit for Gate-Level Logic Synthesis Applications

Siang-Yun Lee
LSI, EPFL
Switzerland

Heinz Riener
Cadence Design Systems
Germany

Giovanni De Micheli
LSI, EPFL
Switzerland

Abstract

Correctness and robustness are essential for logic synthesis applications, but they are often only tested with a limited set of benchmarks. Moreover, when the application fails on a large benchmark, the debugging process may be tedious and time-consuming. In some fields such as compiler construction, automatic testing and debugging tools are well-developed to support developers and provide minimal guarantees on program quality. In this paper, we adapt fuzz testing and delta debugging techniques and specialize them for gate-level netlists commonly used in logic synthesis. Our toolkit improves over similar tools specialized for the AIGER format by supporting other gate-level netlist formats and by allowing a tight integration to provide 10× speed-up. Experimental results show that our fuzzer captures defects in mockturtle, ABC, and LSOracle with 10× smaller testcases and our testcase minimizer extracts minimal failure-inducing cores using 2× fewer oracle calls.

1 Introduction

Logic synthesis is the task of turning an abstract specification into a gate-level netlist composed of logic gates while considering cost functions for area, delay, and power. Logic synthesis plays a crucial role in modern microchip design flows. Sophisticated algorithmic solutions are readily available as open-sourced libraries and tools [4, 21, 25]. These algorithms can optimize netlists with millions of gates within only a few seconds, relying on bit-precise reasoning engines. The inherent complexity of these engines, optimized for many corner cases, makes logic synthesis algorithms susceptible to design and implementation errors. Moreover, algorithms are often only tested on fixed benchmark suites, such as the EPFL logic synthesis benchmarks [1]. Due to numerous possibilities to implement the same Boolean function with different circuit structures, it is not rare that subtle faults slip through the development process and only show themselves when the algorithm is used in practice.

Motivated by the success of automated testing methods, we argue that directed testing approaches and bug-pointing tools specialized for logic synthesis applications can support the developers in detecting bugs earlier, can make implementations more robust, and ultimately lead to a reduction in the time and effort spend for debugging. Due to the large state space and homogeneity of the commonly used netlist formats, general-purpose testing and debugging tools often are incapable of providing the necessary performance to efficiently test implementations of logic synthesis algorithms. The C++ logic network library *mockturtle* [24, 25] has deployed a framework for unit testing, continuous integration on various operating systems and compilers, and a static code analysis engine controlled by user-defined queries to aid developers.

In this paper, we present the latest additions to *mockturtle* for testing and debugging gate-level netlists: (1) a fuzz tester that repeatedly generates small- and intermediate-sized netlists to hunt for bugs and (2) a testcase minimizer to isolate the failure-inducing core of potentially lengthy bug report. The two methods advance existing ones in the AIGER utilities[1] with the following key highlights:

1. Our methods are agnostic of the network type and support different gate-level netlist formats. Although conceptually not hard to implement, to our knowledge this is the first time that automated debugging techniques are available for logic representations such as *Majority Inverter Graphs* (MIGs) or *Xor And Graphs* (XAGs). We demonstrate with a case study in Section 4.1 that testing with more compact representations like XAGs increases the possibility of capturing rare defects.
2. Our implementations are tightly integrated into *mockturtle*, which eliminates interfacing overheads and provides about 10× speed-up over using external testing and debugging solutions.
3. Our fuzz tester provides systematic approaches to test on small circuit topologies in addition to purely-random networks. Experimental results show that our topology-based fuzzer captures defects in *ABC*, *mockturtle* and *LSOracle* using 93% smaller testcases comparing to an existing AIG fuzzer `aigfuzz`[1].
4. Our testcase minimizer guarantees to isolate a minimal failure-inducing core and reduces testcases more efficiently by adopting specialized structural reduction rules for gate-level netlists. Experimental results show that our minimizer isolates smaller or equal-sized cores using 50% oracle calls and 50% runtime comparing to an existing AIG delta debugger `aigdd`[1].

[1] https://github.com/arminbiere/aiger

2 Background

2.1 Scope and Terminologies

Logic networks are technology-independent representations of digital circuits. They model combinational parts of gate-level circuits with directed acyclic graphs, where vertices, or *nodes*, represent logic gates and edges represent interconnecting wires. Prominent examples of logic networks include *And Inverter Graph* (AIG) [13], *Xor And Inverter Graph* (XAG) [11], and *Majority Inverter Graph* (MIG) [2]. Common terms related to structural properties of logic networks, such as *primary input* (PI), *primary output* (PO), *transitive fanin cone* (TFI), *transitive fanout cone* (TFO), and *maximum fanout free cone* (MFFC), are defined the same as in the literature. [3]

This paper focuses on testing and debugging software applications, referred to as the *application under tests* (AUTs), that take a logic network, called a *testcase*, as an input. Prominent examples of such applications include implementations of logic synthesis algorithms such as rewriting [18], resubstitution [17] and technology mapping [6]. Methods to verify the correctness of the results, referred to as the *verification*, are assumed to be provided. They may come from several sources:

- Assertions within the program.
- Memory protection processes in the operating system checking for illegal memory access (typically raising segmentation faults).
- *Combinational equivalence checking* (CEC) [19] of the output network against the input testcase (for logic optimization algorithms).
- Additional code checking coherence of the program's internal data structures, such as checking if the network is acyclic and checking the correctness of reference counts, etc.
- Another algorithm of the same purpose used to provide reference solutions (for problems having an unique correct solution).

A failing verification, e.g., a non-equivalent CEC result, indicates that a *defect* of the AUT is observed and the testcase used is said to be *failure inducing*. The AUT combined with its verification is referred to as an *oracle*, and running the oracle with a testcase is an *oracle call*.

2.2 Fuzz Testing

Fuzz testing [16] is a software testing technique heavily used to detect security-related vulnerabilities and reliability issues. It is conceptually simple, yet empirically powerful. A fuzzing algorithm involves repeatedly generating testcases and using them to test the AUT. The idea of fuzz testing first appeared in 1990, when spurious characters in the command line caused by a noisy dial-up connection to a workstation led to, surprisingly, crashes of the operating system. [16] Nowadays, the generation of testcases in fuzz testing algorithms often involves randomness, and the testcases are supposed to be beyond the expectation of the AUT.

Various taxonomies of fuzz testing algorithms have been developed. For example, black-box fuzzers [14] treat the AUT as a black-box oracle and only observe its input/output behavior, whereas white-box fuzzers [5, 9] analyze some internal information of the AUT and generate testcases accordingly. Depending on the targeted types of AUTs, some fuzzers generate testcases based on predefined models or grammars [8], whereas some other fuzzers mutate an initial *seed* testcase to generate more testcases [7]. There are often some parameters to be set for the testcase generators. A series of fuzz-tests using testcases generated with a specific parameter configuration is called a *fuzz testing campaign*. [15]

2.3 Delta Debugging and Testcase Minimization

Given two versions of the code of a program, where the first version works but the second fails, *delta debugging* [26] is a method originally proposed to extract a minimal set of changes (differences in the two versions of code) that causes the failure. The algorithm was later extended for minimizing failure-inducing testcases. [27]

The basic idea of delta debugging is binary searching and dividing the set of components, may it be the *delta* between two versions of code or the input testcase to a program, testing the program with the reduced set, keeping the subsets that preserve the failure, and increasing the granularity of division. The delta debugging algorithm (*ddmin*) guarantees to find a 1-minimal subset and requires, in the worst case, $n^2 + 3n$ oracle calls, where n is the size of the given set. [27]

Besides delta debugging being a generic method for testcase minimization, researchers have claimed that domain-specific testcase minimization techniques are more effective and efficient for some applications such as tree-structured inputs [20], compilers [22] and SMT solvers [12]. Various open-source implementations of testcase minimization tools exist, including the general-purposed `delta`[2], `aigdd`[1] for the AIGER format, `ddSMT`[3] for the SMT-LIB v2 format, and the LLVM `bugpoint` tool[4]. Inspired by delta debugging, in this paper, we aim at providing such an effective testcase minimization tool specialized for logic networks but not limited to AIGs.

3 Testing and Debugging Toolkit

3.1 Testcase Generation

We develop a fuzz testing framework for testing any application that takes a logic network as input. The AUT and the verification checks are provided as a combined oracle call, thus categorizing it as a black-box fuzzer. Although in some cases of fuzzing, testing with malformed testcases is key to test the robustness of the AUT, this is not the case for our usage. In logic synthesis applications, detecting and rejecting malformed inputs, e.g. a cyclic network, are usually

[2]https://github.com/dsw/delta
[3]http://fmv.jku.at/ddsmt/
[4]https://llvm.org/docs/Bugpoint.html

dealt by the parsers instead of the logic synthesis algorithms. Nevertheless, as logic synthesis applications are often only tested with some common benchmark suites, our fuzzing framework still tests them with a larger input space beyond what they are usually tested with.

To generate random testcases, we propose three parameterized methods. These methods apply to any type of network having a finite set of possible gate types.

Random: Randomly generate nodes in topological order. This method is parameterized by the starting number of PIs n_0, the starting number of gates m_0, the number of networks k of the same configuration to generate, the increment of the number of PIs n and of the number of gates m. The generator starts from generating networks of n n_0 PIs and m m_0 gates and keeps a counter of how many networks have been generated. After generating k networks, the values of n and m are increased by n and m, respectively. Given the current values of n and m, a network is generated by:

1. Create n PIs.
2. Randomly decide on a gate type. Assume that the type requires fanins.
3. Randomly sample nodes (PIs or gates) that have been created.
4. Randomly decide for each fanin if it is complemented.
5. Create the gate. Repeat from step 2 if the number of gates is smaller than m.
6. Assign all nodes without fanout to be POs.

For network types with trivial-case simplifications (e.g., in AIGs, attempting to create an AND gate with identical fanins results in returning the fanin without creating a gate) and structural hashing enabled, the number of gates may not increase after step 4. Thus, the loop of steps 2 to 4 may iterate more than m times and the terminating condition is when the actual number of gates is m. If the parameters are set improperly, e.g., if n 1, this might lead to an infinite loop.

Topology: Exhaustively enumerate all small-sized DAG topologies and randomly concretize them. This method is parameterized by the starting number of gates m_0, the lower r_l and upper r bounds on the PI-to-input ratio and the number of networks k of the same configuration to generate. Upon initialization, the generator enumerates all isomorphic DAG topologies of m m_0 vertices using an algorithm implemented in [10] and randomly shuffles them. Then, it starts from generating networks of the first topology and keeps a counter of how many networks have been generated. After generating k networks, the generator moves on to generating the next topology. After all topologies have been used to generate k networks, the value of m is incremented by 1 and topologies of the increased size are enumerated. Given a topology, which is specified by a DAG G with hanging inputs (i.e., the topology specifies how gates are connected to each other, but not how they are connected to PIs), a random network is concretized by:

1. Let i be the number of hanging inputs in G. Randomly decide on an integer n such that $r_l \quad i \leq n \leq r \quad i$. Create n PIs.
2. For each input of G, randomly decide on a PI to connect to.
3. For each vertex in G, randomly decide on a gate type.
4. For each edge in G, randomly decide whether it is complemented.
5. Assign the last gate to be a PO.

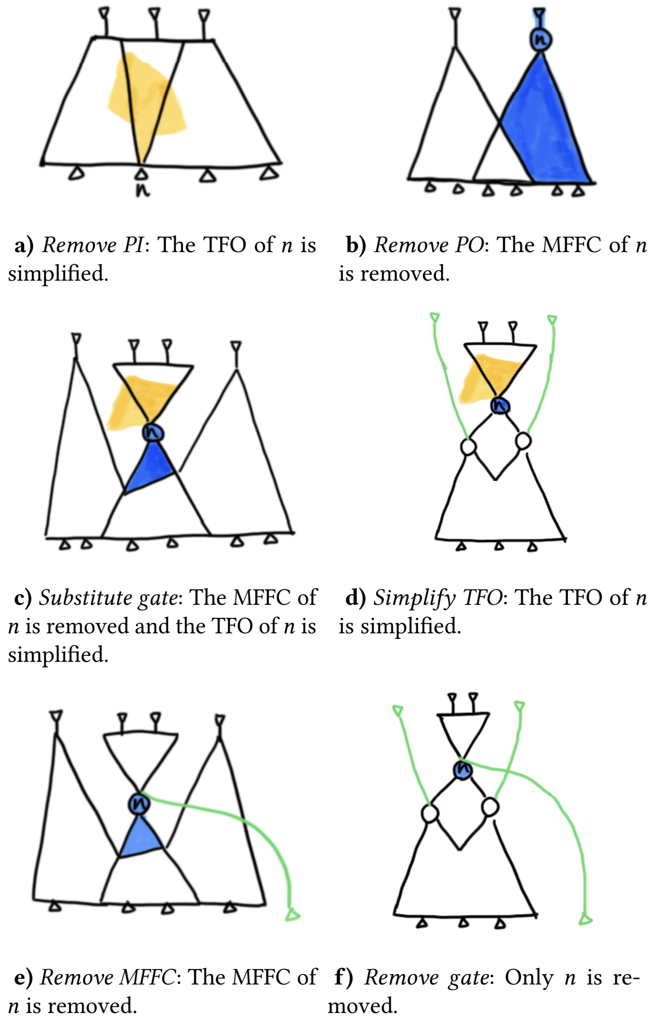
In step 1, lower values of n/i leads to higher probability that the generated network reconverges on PIs, whereas higher values of n/i leads to higher probability to generate a tree-like network. The generated networks are always single output.

Composed: Randomly compose a few small-sized DAG topologies to form a larger network. This method is parameterized by the lower m_l and upper m bounds of the size of DAG topologies, the starting number of components c_0, the starting number of PIs n_0, the number of networks k of the same configuration to generate, the increment of the number of PIs n and of the number of components c. Upon initialization, the generator enumerates all isomorphic DAG topologies of m_l to m vertices. Then, it starts from generating networks of n n_0 PIs and composed of c c_0 components and keeps a counter of how many networks have been generated. After generating k networks, the values of n and c are increased by n and c, respectively. Given the current values of n and c, a network is generated by:

1. Create n PIs.
2. Randomly choose a topology G from the list.
3. For each hanging input of G, randomly decide on an existing node (a PI or a node in a created component) to connect to.
4. For each vertex in G, randomly decide on a gate type.
5. For each edge in G, randomly decide whether it is complemented.
6. If the number of created components is smaller than c, repeat from step 2.
7. Assign all nodes without fanout to be POs.

3.2 Testcase Minimization

Assuming that the concerned defect is deterministic, there is a *core* in any given failure-inducing testcase, which is a subset of the testcase essential for observing the defect. The other parts of the network are said to be *irrelevant* for observing the defect and can be removed. For example, for a defect caused by the algorithm trying to insert an XOR gate into an AIG, which is interpreted as inserting an AND gate instead, a core in the testcase may be a subnetwork computing the XOR function. Due to the localized-computation design style of modern scalable logic synthesis algorithms, the cores are usually small-sized. We say that a core is *minimal* if, for any



a) *Remove PI*: The TFO of n is simplified.

b) *Remove PO*: The MFFC of n is removed.

c) *Substitute gate*: The MFFC of n is removed and the TFO of n is simplified.

d) *Simplify TFO*: The TFO of n is simplified.

e) *Remove MFFC*: The MFFC of n is removed.

f) *Remove gate*: Only n is removed.

Figure 1. Illustration of the reduction stages.

node n, removing n results in never observing the defect again no matter how the fanins and fanouts of n are reconnected. A minimal core in a failure-inducing testcase may or may not be unique. The goal of testcase minimization is to find a minimal core in a given failure-inducing testcase.

We develop a testcase minimization tool for logic networks similar to delta debugging but without adopting binary search. Given a network and an AUT with verification (i.e. an oracle), our testcase minimizer iteratively tries to reduce the network and tests if the defect is still observed. Only the reduction operations that preserve observing the defect are kept; otherwise, the operation is undone. Different reduction operations are tried in six stages with increasing (finer) granularity as follows:

(a) *Remove PI*: Substitute a PI n with constant zero, thus simplifying its TFO by constant propagation. In AIGs, some nodes in the TFO of n that are connected to n without complement are removed, and so are their MFFCs.
(b) *Remove PO*: Substitute a PO n with constant zero, thus removing its MFFC.
(c) *Substitute gate*: Substitute a gate n with constant zero, thus removing its MFFC and simplifying its TFO by constant propagation (as in (a)).
(d) *Simplify TFO*: Assign fanins of a gate n as new POs, and then substitute n with constant zero. This operation is less aggressive than the previous one because only the TFO of n is simplified and its MFFC is kept.
(e) *Remove MFFC*: Substitute a gate n with a new PI. This operation does not cause constant propagation in its TFO and only removes the MFFC of n.
(f) *Remove gate*: Assign fanins of a gate n as new POs, and then substitute n with a new PI or with one of its fanins. Only n is removed.

Figure 1 illustrates the effects of an operation in each of the reduction stages. Regions filled in blue are removed after the operation, and regions marked in yellow are simplified by constant propagation after the operation. Wires and PIs or POs drawn in green are added after the operation.

The relative granularity of stages *remove PI* and *remove PO* depends on the shape of the network. For networks with smaller TFO of PIs and less logic sharing in the TFI of POs, *remove PO* reduces the network faster; for networks with smaller MFFC of POs and more reconvergences near the PIs, *remove PI* reduces the network faster. Thus, the first stage to apply is heuristically decided by whether the network has more PIs than POs (*remove PO* is applied first) or more POs than PIs (*remove PI* is applied first).

In each stage, the minimizer backs-up the current network, randomly samples a PO or a gate as n and performs the corresponding reduction operation. If the defect is not observed anymore after reduction, the back-up is restored. This procedure is repeated until all POs or all gates have been sampled, or until a pre-defined number of operations have been tried.

The resulting network after minimization cannot be reduced anymore because the last stage tries every possibility to remove one gate. Thus, by definition, the minimized testcase is guaranteed to be a minimal core. However, minimal cores are not necessarily unique, so it is possible that a different order of reduction operations (e.g. by using a different random seed) results in a smaller minimal core.

The minimized testcases are, in the most cases, highly destructed and cannot be recognized or reverse-engineered anymore. Therefore, the testcase minimizer does not only facilitate the debugging process, but also the communication between developers when commercially-sensitive benchmarks are involved.

3.3 Usage Example

The testing and debugging toolkit described in this section is implemented in *mockturtle*[5] as part of the EPFL open-source

[5]Available: https://github.com/lsils/mockturtle

logic synthesis libraries [25]. The toolkit supports testing and debugging any application that takes a logic network, written in AIGER (for AIGs) or Verilog (for other network types supported in *mockturtle*, such as XAGs and MIGs) formats, as input.

Figure 2 shows an example workflow of our toolkit. In this example, the toolkit is used to fuzz test an algorithm implemented in *mockturtle* (marked in green), and then, if a defect is observed, minimizes the generated failure-inducing testcase (marked in red). This can be done similarly for other C++-based tools that include *mockturtle* as a library.

```
1   include <mockturtle/mockturtle.hpp>
2  using namespace mockturtle;
3
4  int main()
5  {                                                        Oracle
6    auto opt = [](aig_network aig) -> bool {
7      aig_network const aig_copy = aig.clone();
8      aig_resubstitution(aig);
9      aig_network const miter = *miter(aig_copy, aig);
10     return *equivalence_checking(miter);
11   };
12                                                          Fuzzer
13   fuzz_tester_params fuzz_ps;
14   fuzz_ps.file_format = fuzz_tester_params::aiger;
15   fuzz_ps.filename = "fuzz.aig";
16   fuzz_ps.timeout = 20; // 20 minutes
17   auto gen = random_aig_generator();
18   network_fuzz_tester<aig_network, decltype(gen)>
19     fuzzer(gen, fuzz_ps);
20   bool has_bug = fuzzer.run(opt);
21
22   if ( has_bug) return 0;
23                                                          Minimizer
24   testcase_minimizer_params min_ps;
25   min_ps.file_format = testcase_minimizer_params::aiger;
26   min_ps.init_case = "fuzz";
27   min_ps.minimized_case = "fuzz_min";
28   testcase_minimizer<aig_network> minimizer(min_ps);
29   minimizer.run(opt);
30
31   aig_network aig;
32   lorina::read_aiger("fuzz_min.aig", aiger_reader(aig));
33   write_dot(aig, "fuzz_min.dot");
34   std::system("dot -Tpng -O fuzz_min.dot");
35
36   return 0;
37 }
```

Figure 2. Example code to use the proposed toolkit to generate, minimize, and visualize a failure-inducing testcase.

Our toolkit is also applicable for testing and debugging external tools. In this case, the lambda function in lines 6 to 11 in Figure 2 shall be replaced by one resembles the code in Figure 3.

```
6   auto opt = [](std::string filename) -> std::string {
7     return "abc -c \"read " + filename + "; rewrite\"";
8   };
```

Figure 3. Example code to use the toolkit for testing and debugging an external tool, ABC.

Similar to `aigfuzz` and `aigdd`, calling the oracle as a system command requires switching the program control through the command shell and interfacing the testcases by reading and writing files. With the possibility of a tight integration as in Figure 2, these interfacing overheads can be eliminated and, empirically, making the automated testing and debugging workflow about 10× faster.

4 Case Study

As a case study, we apply the toolkit on a known defect in a variation of *cut rewriting*, which uses a compatibility graph to identify compatible substitution candidates [23], implemented in *mockturtle*.[6] The defect can be observed by having a cyclic network after applying the algorithm. The failure-inducing core of this defect is shown in Figure 4d. The cyclic result is caused by the algorithm observing $n_7 \oplus n_2$ as a substitution for n_{11} and $n_{11} \oplus n_2$ as a substitution for n_7, and trying to apply the two substitutions at the same time. To identify that the two substitution candidates are in conflict, the algorithm should check, for every pair (A, B) of candidates, if the root of A is contained in the cut of B and the root of B is contained in the cut of A. This would be a feasible fix for the defect, but would impact the efficiency of the algorithm. Another rewriting algorithm that does not use the compatibility graph but eagerly substitutes each candidate before searching for the next one is available in *mockturtle*.[7] However, when not affected by the defect, the defective algorithm has on average better quality of result than eager rewriting. Also, the defect seems to be observed very rarely, as will be discussed in Section 4.1. As a compromise, both algorithms are kept in *mockturtle*.

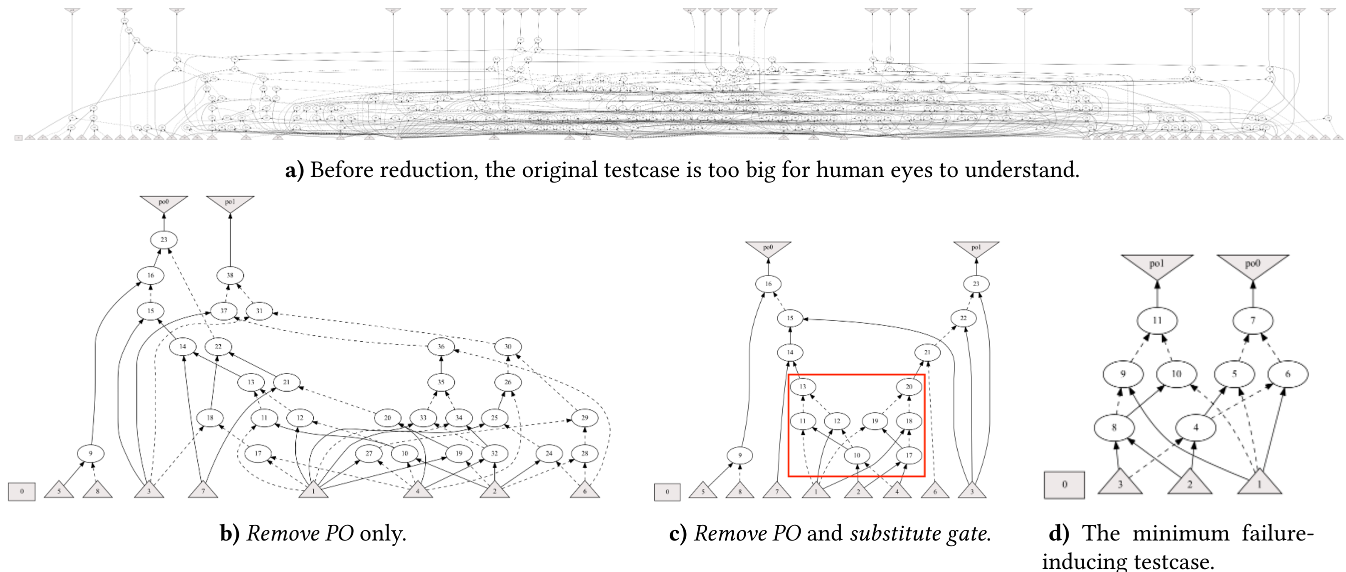
The first reported failure-inducing testcase for this defect is shown in Figure 4a. The original testcase was not minimized by the reporter and have 49 PIs, 272 AND gates, and 28 POs. It took a human expert about 30 minutes to manually reduce the testcase to Figure 4d, with 3 PIs, 8 gates and 2 POs. Using the testcase minimizer, the original testcase is minimized to the same graph (subject to permutations of the two POs) within a second and using 94 oracle calls. In Section 4.2, we study the effectiveness and necessity of the reduction stages described in Section 3.2.

4.1 Capturing The Defect with Fuzz Testing

4.1.1 Using AIGs. Knowing the existence of the defect, we investigate if our fuzz tester is capable of generating another failure-inducing testcase. However, even though the code line coverage has reached its maximum (100% excluding the lines disabled by the algorithm's options), the defect is not observed with more than a billion (10^9) regular (i.e., without leveraging knowledge of the known core) fuzz tests. Even if we limit the sampling space to the 3-input, 8-gate, 2-output topology as in Figure 4d and leaving only the connections to PIs and edge complementations as random choices, there are still $6^2 \times 3^4 \times 2^{16}$ 191 102 976 different possible networks, out of which only 3×2^3 48 networks (equivalent to

[6]The function `cut_rewriting_with_compatibility_graph` can be found in `algorithms/cut_rewriting.hpp`.

[7]The function `cut_rewriting` can be found in the same header file.



a) Before reduction, the original testcase is too big for human eyes to understand.

b) *Remove PO* only.

c) *Remove PO* and *substitute gate*.

d) The minimum failure-inducing testcase.

Figure 4. The failure-inducing testcase for an algorithm implemented in *mockturtle* and intermediate results of minimizing it.

Figure 4d subject to permutation and negation of PIs) are failure-inducing.

This case evidences that rare corner-case defects exist in logic synthesis applications, and the identification of them may only rely on real-world benchmarks. In these cases, the testcase minimization techniques are important to automatize the extraction of the failure-inducing core, which facilitates communication and debugging.

4.1.2 Using XAGs.

We observe that the XOR functions in the core (nodes 9, 10, 11 and nodes 5, 6, 7 in Figure 4d) are necessary. Using any of the randomized methods described in Section 3.1, the possibility of generating an XOR function composed of three AIG nodes is low. However, it is much more likely to generate an XOR gate in an XAG. As the implementation is generic and works for both AIGs and XAGs, we can try to capture the defect using XAGs instead. Table 1 shows that all the three methods successfully capture the defect within reasonable runtime.

Table 1. Fuzzing the defective cut rewriting with XAGs.

Method	#Tests	Time (s)
Random	8150	1.8
Topology	44498	6.6
Composed	77573	22.8

4.2 Effects of The Reduction Stages

Given the initial failure-inducing testcase as in Figure 4a, using the default settings, our testcase minimizer produces the minimal failure-inducing testcase as in Figure 4d, which is a 97% reduction rate in gate count. The minimality can be proved by trying to remove each gate and seeing that any possible resulting testcases are not failure-inducing.

Figures 4b and 4c show the reduction results if only some reduction stages are applied. The first stage, *remove PO* (*remove PI* is skipped because there are more PIs than POs), provides already 89% reduction of the testcase by removing large cones of irrelevant logic and quickly concentrates to the transitive fanin cone of two POs (Figure 4b, 30 gates). The next stage, *substitute gate*, further reduces the size to 15 gates (Figure 4c), and the failure-inducing core is easily observable (marked with a red box). However, the other nodes on top of the core cannot be removed in this stage because substituting any of them with constant zero also removes part of the core. This can be accomplished by adding the stage *simplify TFO*, resulting in Figure 4d. The two key operations are adding PO at nodes 13 and 20 and substituting nodes 14 and 21 with constant zero. It is also possible to reach the minimum by adding only the stage *remove gate*, but it requires at least 6 operations to remove nodes 14, 15, 16, 21, 22 and 23 one by one, showing that this stage operates in a finer granularity. It may seem that the stage *remove MFFC* is not necessary. However, this is only because the failure-inducing core in this example does not have irrelevant transitive fanin gates (i.e., it is connected to PIs) in the original testcase. When this is not the case, the stages *remove MFFC* and/or *remove gate* are necessary to obtain the minimum.

5 Experimental Results

5.1 Fuzzing Open-Source Logic Synthesis Tools

To demonstrate the effectiveness of fuzzing and compare different testcase generation methods, we fuzz-tested the following open-source logic synthesis tools: *mockturtle*[8] [24], *ABC*[9] [4], and *LSOracle*[10] [21]. Table 2 lists the commands

[8]Available: https://github.com/lsils/mockturtle. Commit `cf4769f`.

[9]Available: https://github.com/berkeley-abc/abc. Commit 31519bd.

[10]Available: https://github.com/lnis-uofu/LSOracle. Pull request #81.

Table 2. Fuzz testing results.

	aigfuzz #Tests = 1000			Random #Tests = 1000			Topology #Tests = 5000			Composed #Tests = 5000		
AUT	#FITs	Size	Time	#FITs	Size	Time	#FITs	Size	Time	#FITs	Size	Time
`mockturtle::aig_resubstitution`	0	-	10.3	0	-	3.5	0	-	0.02	1	14.0	0.03
`mockturtle::sim_resubstitution`	3	812.3	22.3	0	-	4.2	6	5.0	0.06	93	21.7	0.11
`abc> bms_start; if -u; strash`	952	2476.0	32.5	1000	950.0	14.6	1716	4.7	12.8	3749	20.3	13.0
`abc> &if; &mfs -dael; &st`	956	2515.1	4.5	969	978.8	1.6	0	-	14.2	1047	23.9	12.3
`abc> &mfsd; &st`	473	3935.9	30.5	584	1078.4	14.1	0	-	14.1	120	24.2	14.1
`abc> &mfsd -cd; &st`	481	3959.7	130.0	73	1266.4	47.8	0	-	14.2	1	20.0	14.4
`abc> if; mfse; strash`	458	2763.3	14.7	93	940.3	13.6	1	5.0	16.0	1056	23.4	15.0
`abc> &stochsyn resub`	13	1164.8	14.6	0	-	6.0	0	-	12.5	14	18.9	12.5
`lsoracle> aigscript`	1	7364.0	38.1	0	-	16.4	0	-	32.8	1	14.0	32.8
`lsoracle> deep`	12	3227.3	41.6	0	-	21.5	0	-	33.3	6	23.0	32.8
`lsoracle> xmgscript`	3	1056.0	21.6	2	350.0	10.5	38	4.9	11.5	99	20.1	12.0
Average		2941.6	32.8		995.5	14.0		4.7	14.7		21.5	14.5

or functions where defects have been observed. Fuzz testing campaigns were conducted on each AUT using aigfuzz and the three network generation methods described in Section 3.1. In each campaign, aigfuzz and the method Random ran 1000 tests, whereas the methods Topology and Composed ran 5000 tests. In Table 2, column #FITs lists the total number of failure-inducing testcases generated, column Size lists the average size (number of gates) of the failure-inducing testscases, and column Time lists the total runtime in minutes including the oracle calls.

The Composed method captured defects in all of the listed AUTs. On average, Composed is about 2× faster than aigfuzz and it tests on 5× more networks. This is because the Composed testcases are, on average, 7% in size comparing to those generated by aigfuzz. Also, notice that for AUTs in *mockturtle*, the runtimes of our fuzzing methods are about 10× faster than aigfuzz thanks to the tight integration.

5.2 Testcase Minimization

We compare our testcase minimizer to aigdd using the user-reported failure-inducing testcase in Section 4 and four bigger testcases found by fuzz testing in Section 5.1. In Table 3, column Size lists the number of gates of the original and the minimized testcases, column #Calls lists the number of oracle calls and column Time lists the total runtime in seconds. It can be observed that our minimizer reduces the testcases into minimal cores of roughly the same or smaller sizes comparing to aigdd, using on average 50% oracle calls and 50% runtime.

6 Conclusion and Discussion

In this paper, we survey automated testing and debugging techniques and provide an open-sourced toolkit specialized for gate-level logic synthesis applications. While random fuzz testing can already catch many higher-frequency defects, the topology-based fuzzing methods provide a more systematic approach to thoroughly test topology-related corner cases. After failure-inducing testcases are found, the testcase minimizer can be used to reduce their size efficiently to facilitate manual debugging (and also anonymizing sensitive testcases). Moreover, our testcase minimization technique guarantees to find a minimal core in the failure-inducing testcase, which often gives insights on the cause of the defect and may also be used to categorize testcases for the same AUT. The case study shows that (1) some defects may be difficult to catch by fuzz testing, thus testcase minimization is important when we need to rely on real-world testcases; and (2) testing with more functionally-compact networks, such as XAGs, may help to detect some defects in generic logic synthesis algorithms. In the remaining of this section, we discuss more potentials of the toolkit and directions for future research.

6.1 Non-deterministic Defects

Non-deterministic defects may be hard to debug because they cannot always be reproduced. Non-determinism may come from a random number generator without a fixed seed, a race condition in concurrent computation, or accessing to uninitialized or unintended (index-out-of-bounds) memory. If a non-deterministic defect is first observed with a large testcase, it may be difficult to minimize it while maintaining the defect being observed. In such cases, fuzz testing may help generating smaller testcases to observe the defect deterministically.

6.2 Other Applications of The Toolkit

In addition to testing and debugging, the proposed tools can also be used for finding examples with specific properties. For example, an open problem in logic synthesis is whether it is better to heavily optimize an AIG before transforming into MIG, or to perform optimization directly with an MIG. Our toolkit can be used to generate minimal examples where

Table 3. Testcase minimization results.

AUT	Original size	aigdd			Ours		
		Size	#Calls	Time	Size	#Calls	Time
`mockturtle::cut_rewriting_with_compatibility_graph`	272	8	210	32.5	8	96	0.1
`mockturtle::sim_resubstitution`	615	7	735	11.6	8	351	2.5
`abc> &mfsd -cd; &st`	1050	31	1198	150.0	20	857	120.5
`abc> if; mfse; strash`	1850	6	834	61.2	5	333	30.3
`abc> &stochsyn resub`	3228	10	1124	59.6	8	411	21.1

one optimization script obtains better results than the other, which might help researchers identify weaknesses in the algorithms.

6.3 Future Works

Our network fuzzer currently does not support generating k-LUT networks easily without specifying all possible LUT functions as different gate types. This can be mitigated by integrating a random truth table generator.

In addition to minimizing the failure-inducing input networks, when the defective AUT involves multiple independent algorithms (i.e., a *script* with a sequence of *commands*), it would also be helpful to minimize the script and remove irrelevant commands. This can be accomplished by automatic binary search, similar to delta debugging.

Acknowledgments

We thank Max Austin for providing the original testcase discussed in Section 4.

References

[1] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. The EPFL combinational benchmark suite. In *IWLS 2015*.

[2] Luca Amaru, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. Majority-inverter graph: A new paradigm for logic optimization. *IEEE TCAD* 35, 5 (2015), 806–819.

[3] Robert Brayton and Alan Mishchenko. 2006. Scalable logic synthesis using a simple circuit structure. In *IWLS 2006*, Vol. 6. 15–22.

[4] Robert K. Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *CAV 2010*. 24–40.

[5] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI 2008*. USENIX Association, 209–224.

[6] Alessandro Tempia Calvino, Heinz Riener, Shubham Rai, Akash Kumar, and Giovanni De Micheli. 2022. A Versatile Mapping Approach for Technology Mapping and Graph Optimization. In *ASP DAC 2022*. IEEE, 410–416.

[7] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *SP 2015*. IEEE Computer Society, 725–741.

[8] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language fuzzing using constraint logic programming. In *ASE 2014*. ACM, 725–730.

[9] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS 2008*. The Internet Society.

[10] Winston Haaswijk, Mathias Soeken, Alan Mishchenko, and Giovanni De Micheli. 2020. SAT-Based Exact Synthesis: Encodings, Topology Families, and Parallelism. *IEEE TCAD* 39, 4 (2020), 871–884.

[11] Ivo Hálecek, Petr Fiser, and Jan Schmidt. 2017. Are XORs in logic synthesis really necessary?. In *DDECS 2017*. 134–139.

[12] Gereon Kremer, Aina Niemetz, and Mathias Preiner. 2021. ddSMT 2.0: Better Delta Debugging for the SMT-LIBv2 Language and Friends. In *CAV 2021*. Springer, 231–242.

[13] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. 2002. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE TCAD* 21, 12 (2002), 1377–1394.

[14] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip A. Porras. 2017. DELTA: A Security Assessment Framework for Software-Defined Networks. In *NDSS 2017*. The Internet Society.

[15] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Trans. Software Eng.* 47, 11 (2021), 2312–2331.

[16] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[17] Alan Mishchenko, Robert K. Brayton, Jie-Hong R. Jiang, and Stephen Jang. 2011. Scalable don't-care-based logic optimization and resynthesis. *ACM Trans. Reconfigurable Technol. Syst.* 4, 4 (2011), 34:1–34:23.

[18] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. 2006. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *DAC 2006*. 532–535.

[19] Alan Mishchenko, Satrajit Chatterjee, Robert K. Brayton, and Niklas Eén. 2006. Improvements to combinational equivalence checking. In *ICCAD 2006*. ACM, 836–843.

[20] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *ICSE 2006*. ACM, 142–151.

[21] Walter Lau Neto, Max Austin, Scott Temple, Luca G. Amarù, Xifan Tang, and Pierre-Emmanuel Gaillardon. 2019. LSOracle: a Logic Synthesis Framework Driven by Artificial Intelligence: Invited Paper. In *ICCAD 2019*. ACM, 1–6.

[22] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *PLDI 2012*. ACM, 335–346.

[23] Heinz Riener, Winston Haaswijk, Alan Mishchenko, Giovanni De Micheli, and Mathias Soeken. 2019. On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis. In *DATE 2019*. IEEE, 1649–1654.

[24] Heinz Riener, Eleonora Testa, Winston Haaswijk, Alan Mishchenko, Luca G. Amarù, Giovanni De Micheli, and Mathias Soeken. 2019. Scalable Generic Logic Synthesis: One Approach to Rule Them All. In *DAC 2019*. ACM, 70.

[25] Mathias Soeken, Heinz Riener, Winston Haaswijk, Eleonora Testa, Bruno Schmitt, Giulia Meuli, Fereshte Mozafari, and Giovanni De Micheli. 2018. The EPFL logic synthesis libraries. *arXiv preprint arXiv:1805.05121* (2018).

[26] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *SIGSOFT 1999*. Springer, 253–267.

[27] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.