CirFix: Automatically Repairing Defects in Hardware Design Code

Hammad Ahmad
hammada@umich.edu
University of Michigan, Ann Arbor
Ann Arbor, Michigan, USA

Yu Huang
yhhy@umich.edu
University of Michigan, Ann Arbor
Ann Arbor, Michigan, USA

Westley Weimer
weimerw@umich.edu
University of Michigan, Ann Arbor
Ann Arbor, Michigan, USA

ABSTRACT

This paper presents CirFix, a framework for automatically repairing defects in hardware designs implemented in languages like Verilog. We propose a novel fault localization approach based on assignments to wires and registers, and a fitness function tailored to the hardware domain to bridge the gap between software-level automated program repair and hardware descriptions. We also present a benchmark suite of 32 defect scenarios corresponding to a variety of hardware projects. Overall, CirFix produces plausible repairs for 21/32 and correct repairs for 16/32 of the defect scenarios. This repair rate is comparable to that of successful program repair approaches for software, indicating CirFix is effective at bringing over the benefits of automated program repair to the hardware domain for the first time.

CCS CONCEPTS

• **Hardware → High-level and register-transfer level synthesis**; **Bug fixing (hardware)**; • **Software and its engineering → Search-based software engineering**.

KEYWORDS

automated program repair, hardware designs, HDL benchmark

ACM Reference Format:
Hammad Ahmad, Yu Huang, and Westley Weimer. 2022. CirFix: Automatically Repairing Defects in Hardware Design Code. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages.
https://doi.org/10.1145/3503222.3507763

1 INTRODUCTION

Recent increases in the complexity of hardware designs have challenged the ability of developers to find and repair defects in circuit descriptions [68]. While significant effort has been devoted to efficiently verifying functional correctness in hardware design descriptions, relatively little work has been done in patching defects in such descriptions automatically. By and large, debugging and repairing hardware designs remains a very expensive and time-consuming task [20]. Indeed, recent functional and security vulnerabilities due to defects at the hardware design level have led to expensive consequences [8, 43, 83]. To reduce the cost and improve the maintenance of hardware designs, a solution needs to not only precisely identify sources of defects in real-world off-the-shelf hardware descriptions, but also automatically produce repairs implementing correct functionality of the circuit designs that can then be shown to developers for validation before moving on to the synthesis phase. Additionally, we desire a solution that applies directly to both the behavioral aspects (i.e., higher-level descriptions of circuit functionality) and the register-transfer level (RTL) aspects (i.e., lower-level descriptions) of circuit designs, and makes use of readily-available resources that are part of hardware design to validate proposed repairs.

Previous work has attempted to address this problem but may not satisfy all of these characteristics of a desired solution. For instance, some techniques automatically localize defects in design source code but suffer from high false positive rates [29, 65]. Other approaches for automatic error diagnosis and correction require formal specifications to conduct design verification [12], which usually do not scale to large designs. Furthermore, previous work does not operate on behavioral-level descriptions of hardware circuits [13, 49]. On the other hand, in the realm of software, significant research effort focuses on repairing bugs automatically [21, 46, 58]. *Automated program repair* (APR) algorithms fix defects in software by producing patches that pass all test cases while retaining required functionality. Traditional APR for software employs *fault localization* techniques to implicate faulty code, and such techniques are often crucial to the success of program repair.

While both software programs and hardware description languages (HDLs) share programming concepts like expressions, statements, and control structures, suggesting the possibility of repurposing software repair techniques to hardware designs, we highlight two key differences between the two domains: (1) Software programs are typically based around a serial execution model, where one line of code executes before the next. By contrast, HDL designs are inherently parallel and often include non-sequential statements, since separate portions of hardware can operate simultaneously. (2) Software programs usually use test cases to evaluate functional correctness, where individual test cases may pass or fail depending on the quality of the software. HDL designs, on the other hand, use *testbenches* [50], which are programs with documented and repeatable sets of stimuli, to simulate behaviors of a device under test (DUT). In both academia and industry, the majority of digital hardware design is done using such HDLs.

We present two key insights to bridge the gap between well-established software repair techniques and hardware designs. We first hypothesize that while traditional spectrum-based fault localization approaches do not apply to hardware designs that feature

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507763

a more parallel structure [26], dataflow-based fault localization (e.g., [5]) approaches work well in this domain. Second, we hypothesize that a traditional hardware testbench can be instrumented to admit observations for candidate patches that guide the search for APR.

Leveraging these insights, we present CirFix, a framework for automatically repairing defects in hardware designs implemented in languages like Verilog, one of the most popular HDLs [34]. CirFix uses genetic programming (GP), an iterative stochastic search technique, to find candidate repairs for defects in hardware designs. CirFix also makes use of readily-available artifacts in the hardware design process (e.g., testbenches, simulation environments) to diagnose and repair defects in a circuit description. We propose an approach to guide the search for a repair by instrumenting hardware testbenches to record the values of output wires at specified time intervals during a simulation of the circuit design. CirFix then performs a bit-level comparison of output wires against information for expected behavior to assess functional correctness of candidate repairs. CirFix employs a fixed point analysis of assignments made to internal registers and output wires to implicate statements and reduce the search space, enabling our approach to scale to large circuit designs in industry.

We also present a benchmark suite of 32 *defect scenarios* [39] based on three hardware experts — two from industry and one from academia — asked to transplant bugs they observed in real life into 11 different Verilog projects. CirFix can produce plausible repairs for 21 out of the 32 Verilog defect scenarios within reasonable resource bounds, of which 16 are deemed correct upon manual inspection.

The main contributions of this paper are:

- CirFix, a hardware-design automated repair algorithm.
- A novel dataflow-based fault localization approach for HDL descriptions to implicate faulty design code.
- A novel approach to guide the search for a hardware design repair that is compatible with the testbench-based hardware testing process.
- A new benchmark suite of 32 scenarios, based on proprietary bugs but available in 11 open Verilog projects.
- A systematic evaluation of CirFix on our benchmark suite. CirFix was able to correctly repair 16 out of the 32 Verilog defects under consideration.

2 MOTIVATING EXAMPLE

In this section, we use an example defect from a faulty 4-bit counter with an overflow bit, implemented in Verilog, to motivate the fault localization and candidate evaluation approaches used by CirFix. The main block of the source code is shown in Figure 1a, with the corresponding testbench in Figure 1b. The circuit design uses wires `enable` and `reset` to increment (lines 35–37) and reset (lines 30–33) the counter respectively. Incrementing the counter when it has a binary value of `4'b1111` results in the overflow bit being set to true (lines 39–41). This implementation incorrectly manages the overflow bit: the if-statement at line 30 is missing an assignment that resets `overflow_out`. Such defects can have serious consequences — integer overflow errors can be leveraged into significant security exploits [14].

```
27  always@(posedge clk) // Execute at each rising edge of the clock signal
28  begin: COUNTER
29      // If reset is active, reset the outputs to 0
30      if(reset==1'b1) begin
31          counter_out <= #1 4'b0000;
32          // Missing: overflow_out <= #1 1'b0;
33      end
34      // If enable is active, increment the counter
35      else if(enable == 1'b1) begin
36          counter_out <= #1 counter_out + 1;
37      end
38      // If the counter overflows, set overflow_out to be 1
39      if(counter_out == 4'b1111) begin
40          overflow_out <= #1 1'b1;
41      end
42  end
```

(a) Main block of the 4-bit counter with an overflow error

```
50  always #5 clk = !clk; // Set clock signal oscillations
51
52  initial begin // Execute this block once
53      #5 // Wait for 5 time units
54      forever begin // Execute this block indefinitely until simulation stops
55          @(reset_trigger); // Wait for the reset_trigger event
56          @(negedge clk);
57          reset = 1; // Set reset to 1 on the next falling edge of the clock
58          @(negedge clk);
59          reset = 0; // Set reset to 0 on the next falling edge of the clock
60          -> reset_done_trigger; // Send the reset_done_trigger event signal
61      end
62  end
63
64  initial begin
65      #10 -> reset_trigger; // Send the reset_trigger event signal after 10 time units
66      @(reset_done_trigger); // Wait for the reset_done_trigger event
67      @(negedge clk); // Wait for falling edge of the clock signal
68      enable = 1; // Enable the counter
69      repeat (21) begin // Wait for 21 more falling edges of the clock signal
70          @(negedge clk);
71      end
72      enable = 0; // Disable counter
73      #5 -> terminate_sim; // Terminate simulation after 5 time units
74  end
```

(b) Main testing logic from the 4-bit counter testbench

Figure 1: A 4-bit counter with an overflow error in Verilog.

For the purposes of this work, there are two key hardware design concepts that we highlight for a general audience: circuit synchronization and parallelism.

Circuit synchronization. The main block of the circuit design code shows an *always* block (line 27, Figure 1a) that executes repeatedly until the simulation stops. The execution of such blocks can only be triggered by changes to wires in the *sensitivity list* that follows the `always` keyword. Nearly every digital circuit design includes a *clock signal* (line 50, Figure 1b) that oscillates between a high and a low state (denoted by events `posedge` and `negedge` respectively); circuits rely on clock signals to know when and/or how to execute their programmed actions. A *clock cycle* is the period of time it takes for the clock signal to oscillate from high to low and back to a high state. For the 4-bit counter in Figure 1a, the wire `clk` (denoting the clock signal) is the only wire in the always block's sensitivity list (see line 27), and lines 28–42 are executed every time that wire reaches a high state.

Parallelism. A key property of HDL designs not immediately apparent in Figure 1 is that parts of the design code typically execute in parallel. When a design is realized into actual hardware, individual components run all the time. Indeed, every statement in a Verilog design not inside an explicit sequential block of code exhibits concurrency. For instance, for the 4-bit counter in Figure 1a, an implementation managing the overflow bit correctly would include two assignments to `counter_out` and `overflow_out` (on lines 31

and 32 respectively) that happen at the same time when `reset` is true.

To automatically repair the design code in Figure 1a, CirFix needs to first answer, for the original design and each candidate repair: *what part of the circuit, if any, is behaving incorrectly?* Unfortunately, standard spectrum-based fault localization tools commonly used by APR for software do not work for HDL designs that exhibit parallelism. To overcome this challenge, we propose a novel fault localization approach based on assignments to wires and registers. We first instrument the existing testbench to record output values at given time intervals. This instrumented testbench, when used to simulate the design, reports the output values from the circuit, which can be compared against expected output. Any mismatch between expected and actual output serves as the starting point for our fault localization. For the 4-bit counter in Figure 1, the testbench waits for 10 units of time before sending the reset signal (line 65, Figure 1b — cf. stimuli for unit tests in software). The procedural block within the testbench that was waiting on the reset signal (line 55, Figure 1b) then sets `reset` to true upon the next falling edge of the clock signal. This causes any subsequent executions of the corresponding if-statement that resets the wires (line 30, Figure 1a) to evaluate the true branch, following which the counter is reset. A correct design should also reset the overflow bit: at this point, the expected output for the circuit requires `overflow_out` to be 0, while the actual value recorded by our instrumented testbench is *x* (the Verilog representation an uninitialized or unknown logic value). This causes `overflow_out` to be implicated for fault localization, and CirFix focuses repair efforts on assignments to this wire and parts of design code that such assignments transitively depend on (e.g., the conditional in line 39, Figure 1a).

For every candidate repair produced, CirFix needs to also answer: *how good (i.e., fit) is the proposed repair at fixing the defect?* Unfortunately, evaluation approaches for candidate repairs from software cannot be applied to HDL descriptions that typically use testbenches (see Figure 1b). We address this using a novel fitness evaluation approach. Our instrumented testbench records the values of output wires and registers at every rising edge of the clock during an otherwise standard hardware simulation. For developer-specified time intervals from the design simulation (a clock cycle by default), our *fitness function* compares each output bit against the expected output: for every bit match, we add to the fitness sum; for every bit mismatch, we subtract from the sum. This fitness sum is then normalized. For the 4-bit counter shown in Figure 1, the testbench simulates the design code for 26 clock cycles, out of which the first 20 produce an output of *x* (i.e., uninitialized) for `overflow_out` on the original design. This causes an output mismatch for `overflow_out` for 17 clock cycles, resulting in a fitness score of 0.58 (see Section 3.2 for CirFix fitness calculations). A repair managing `overflow_out` correctly would match expected behavior, resulting in a fitness of 1.0.

This faulty circuit code was obtained by having a hardware expert from industry adversarially transplant defects from their experience into open circuit descriptions (see Section 4). We use this example to motivate and demonstrate the basic design ideas behind CirFix, an approach that scales well to larger circuit designs, as we will demonstrate.

Algorithm 1 The high-level CirFix pseudocode.

Input: Circuit design to be repaired, C.
Input: Instrumented testbench for circuit, *TB*.
Input: Expected output for circuit behavior, O.
Input: Fitness function, f.
Input: Parameters, *popnSize*, *maxGens*, *rtThreshold*, *mutThreshold*.
Output: Repaired circuit description.

```
1:  popn ← seed_popn(C, popnSize)
2:  repeat
3:      childPopn ← ∅
4:      while |childPopn| ≤ popnSize and
        ∀ candidate ∈ childPopn. f(candidate, TB, O) < 1.0 do
5:          parent ← tournament_selection(popn, f)
6:          fl_set ← fault_loc(parent)
7:          if probability() ≤ rtThreshold then ▷ Repair templates
8:              child ← apply_fix_pattern(parent, fl_set)
9:              childPopn ← childPopn ∪ {child}
10:         else                                ▷ Repair operators
11:             if probability() ≤ mutThreshold then
12:                 child ← mutate(parent, fl_set)
13:                 childPopn ← childPopn ∪ {child}
14:             else
15:                 parent2 ← tournament_selection(popn, f)
16:                 {child1, child2} ← crossover(parent, parent2)
17:                 childPopn ← childPopn ∪ {child1, child2}
18: until resources exhausted or
    ∃ candidate ∈ childPopn. f(candidate, TB, O) = 1.0
19: return minimize(candidate, TB, O)
```

3 TECHNICAL APPROACH

In this section, we present CirFix, an automated repair algorithm for defects in hardware design code. Our prototype implementation of CirFix operates on hardware descriptions written in Verilog. The pseudocode for the main CirFix loop is shown in Algorithm 1.

CirFix applies our two-pronged HDL-specific approach to implicate faulty design code and assess the correctness of circuit descriptions to produce repairs that can then be shown to human developers for review. Our *fault localization* approach simulates a faulty circuit and assigns blame to incorrect wire and register outputs (line 6 in Algorithm 1; see Section 3.1). Note that while traditional software-based APR techniques typically compute fault localization once at the start of the search for repairs, we choose to repeatedly re-localize to support multiple dependent edits made to the source code. Our *fitness function*, tailored to the hardware domain, assigns scores to each candidate patch to guide the search for repairs (lines 4 and 18 in Algorithm 1; see Section 3.2).

At a high level, CirFix uses genetic programming (GP) [36], an iterative stochastic search technique, to synthesize candidate repairs to faulty HDL programs. The framework takes as input the source code implementing a faulty circuit design, an instrumented testbench used to simulate the circuit for testing and verification purposes, the expected circuit behavior,[1] and the input parameters. The algorithm starts with the original source code and maintains a

[1]Note that CirFix does not require perfect information for expected behavior for every timestep: the developer can choose to only provide information at certain time

Algorithm 2 High-level algorithm for fault localization for HDL based on a fixed point analysis of assignments.

Input: Faulty circuit design code AST, ast.
Input: Simulation output, $S : Time \rightarrow Var \rightarrow \{0, 1, x, z\}$.
Input: Expected output, $O : Time \rightarrow Var \rightarrow \{0, 1, x, z\}$.
Output: Fault localization set, FL.

```
1:  FL, mismatch ← ∅, ∅
2:  mismatch′ ← get_output_mismatch(O, S)          ▷ Section 3.2
3:  while mismatch ≠ mismatch′ do                  ▷ Fixed point calculation
4:      mismatch ← mismatch ∪ mismatch′
5:      for node in ast do
6:          if implicated(node, mismatch) then
7:              FL ← FL ∪ {node.id}
8:              for each child of node do
9:                  FL ← FL ∪ {child.id}
10:                 if type(child) = Identifier and
    child.name ∉ mismatch then
11:                     mismatch′ ← mismatch′ ∪ {child.name}
12: return FL
```

population of program variants, each stored as a *repair patch* [2] describing a sequence of abstract syntax tree (AST) edits parameterized by unique node numbers. Each program variant is obtained by applying a *repair operator* (lines 12 and 16 in Algorithm 1; see Section 3.4) or a *repair template* (line 8 in Algorithm 1; see Section 3.3) to a parent selected for reproduction. Candidate variants are selected for reproduction based on their *fitness* scores assigned by the CirFix fitness function (line 5 in Algorithm 1; see Section 3.5). Our *fix localization* identifies code to be inserted or replaced as part of mutation operations (see Section 3.6). The algorithm loops for several *generations*, each maintaining a population of program variants, until a *plausible repair* is found that produces output (as observed by the instrumented testbench) matching the expected circuit output, or allowed resources are exhausted (i.e., the algorithm reaches a timeout or a certain number of generations). For the final post processing step, CirFix *minimizes* [87] a candidate repair to remove extraneous operations not needed to obtain correct circuit output (line 19 in Algorithm 1; see Section 3.7). Candidate repairs are not deployed directly but are instead shown to human developers (e.g., during the pair process between an RTL design engineer and a verification engineer [10]) for validation before the design is ultimately synthesized, reducing maintenance costs [48, 84].

3.1 Fault Localization

Fault localization is critical to the success and efficiency of APR [40]. Traditional APR for software often relies on spectrum-based fault localization [31] to narrow down defects to certain parts of a faulty program by sampling the program counter. Such fault localization approaches do not extend naturally to the parallel structure of hardware descriptions [26].

To overcome this challenge, we propose a novel dataflow-based fault localization approach to implicate faulty code in HDL descriptions. Previous work analyzing defects in large hardware projects reports that most defects in Verilog descriptions correspond to assignment statements and if-statements [75]. We present an algorithm that implements an analysis of assignments made to wires and registers in a circuit's design code to implicate faulty statements. Our proposed algorithm transitively captures data and control dependencies in a context-insensitive fixed point analysis. While traditional spectrum-based fault localization approaches for software return a ranked list of implicated statements [1, 30, 60], our approach returns a uniformly-ranked set: due to the parallel structure of HDL designs, a set of implicated assignments that are equally likely to contribute to the design defect suffices.

Algorithm 2 outlines the high-level pseudocode for our fault localization approach. The algorithm takes as input the AST of the faulty circuit design, the output from design simulation, and the expected circuit behavior (see Section 3.2 for the structure of the simulated and expected outputs). It then compares the simulation output against the expected behavior to produce a set of *identifier* names (i.e., variable names) for output wires and registers with mismatched values. Using this mismatch set as a starting point, for every node in the AST, the algorithm checks if the node is implicated by output mismatch. Implication for a node in the AST occurs when

- (Impl-Data): either the node corresponds to an assignment statement and the left child of the node corresponds to an identifier in the mismatch set (cf. data dependency analysis),
- (Impl-Ctrl): or the node corresponds to a conditional statement and an identifier in the conditional statement belongs to the mismatch set (cf. control dependency analysis).

Any implicated node and all of the node's children are added to the fault localization set. Additionally, if any child of an implicated node is itself an identifier not part of the mismatch set, the name of the identifier is added to the mismatch set (Add-Child). For example, for the 4-bit counter introduced in Section 2, recall that the `overflow_out` wire had incorrect output from the circuit simulation. This causes the wire to be added to the mismatch set. The CirFix fault localization implicates the only assignment to `overflow_out` (line 40, Figure 1a) by rule (Impl-Data) in the first iteration of the algorithm. Indeed, the entire if-statement wrapping this assignment (line 39, Figure 1a) gets implicated by (Impl-Ctrl), which brings in the new identifier `counter_out` to the mismatch set by (Add-Child). The process is repeated until there are no new identifiers added to the mismatch set, following which the fault localization set is output.

This novel approach to fault localization for hardware is a good fit for automatically repairing HDL designs: it returns a precise set of implicated AST nodes in a faulty circuit design, is context-insensitive and therefore inexpensive to compute, and applies directly to node types associated with ASTs for languages like Verilog.

3.2 Fitness Evaluation

The *fitness function* evaluates the acceptability of a program variant by assigning a value ranging continuously between 0 and 1 to the variant, with 1 indicating a *plausible* [64] (i.e., testbench-adequate) repair ready to be shown to human developers. Fitness provides a termination criterion for CirFix and guides the search for a repair. As mention in Section 1, traditional APR for software uses test-case

intervals. See Section 5.4 for an evaluation of the trade-off between the level of detail of expected output and repair success.

Simulation Result

```
time, ..., overflow_out
. ..
25,   ..., x
35,   ..., x
45,   ..., x
...
165,  ..., x
175,  ..., x
185,  ..., x
195,  ..., x
205,  ..., 1
...
```

Expected Behavior

```
time, ..., overflow_out
. ..
25,   ..., x
35,   ..., 0
45,   ..., 0
...
165,  ..., 0
175,  ..., 0
185,  ..., 0
195,  ..., 0
205,  ..., 1
...
```

Figure 2: A comparison between the simulation result and the expected behavior information for the faulty 4-bit counter in the motivating example. Wires with the correct output are omitted for space reasons. Note the output mismatch for the `overflow_out` wire for timestamps 35 through 195.

based evaluation strategies to assess candidate repairs. Hardware designs, by contrast, use testbenches to verify functional correctness. We present a novel fitness function tailored to hardware to guide the search for repairs to HDL designs. Our fitness function uses two key insights: *visibility* and *comparison*.

Traditional hardware testbenches monitor the values of output wires during simulation and assess correctness based on the final output values. For instance, the testbench for the 4-bit counter introduced earlier (Figure 1b) may report that the final value of the counter is 5 and the overflow bit is 1 when the simulation terminates. Some off-the-shelf hardware testbenches, especially those for large projects, may not even report the exact incorrect value, reporting instead merely the presence or absence of an error during simulation. We want our fitness function to assess a candidate repair based on intermediary as well as final output values, and assign fitness values to the repair based on its overall closeness to the correct circuit design [32]. To do so, given a testbench for a faulty HDL description, we instrument the testbench to record the values of output wires and registers for specified time intervals. This instrumentation is easily automatable: every hardware testbench must instantiate a device-under-test (DUT) and connect wires to the module being instantiated (cf. unit tests in software instantiating the object being tested); each module in turn specifies input and output wires, and a static analysis of the instantiation of the DUT can provide the information needed to instrument a testbench automatically.

Once the testbench is instrumented, we simulate the circuit design and compare the results against the expected output to assess functional correctness of the HDL description. We desire a fitness function that assigns high values to candidate repairs that display behavior similar to expected behavior. To do so, we need to determine the relative contribution of each bit to the fitness of a proposed repair. Given a set of time steps $Time$, a set of output wires and registers Var, a simulation result $S : Time \rightarrow Var \rightarrow \{0, 1, x, z\}$, and expected output $O : Time \rightarrow Var \rightarrow \{0, 1, x, z\}$, where x or z correspond to unknown logic value and high impedance respectively, for timestamp $c_i \in Time$, we sum over the $n = |S(c_i)|$ output bits of the circuit. We compare the expected value for wire b from clock cycle c_i, $O_{c_i,b} = O(c_i(b))$, against the actual value from the simulation result, $S_{c_i,b} = S(c_i(b))$ (see Figure 2 for an indicative example juxtaposing a simulation result with expected behavior). If the bits match, we add to the fitness sum of the circuit; if the bits differ, we subtract from the fitness. An additional penalty weight φ is assigned to bits with values of x (uninitialized) or z (high impedance).

The fitness sum, $sum(S, O)$, and total possible fitness, $total(S, O)$, are defined as follows, where _ represents a bit value of 0 or 1:

$$sum(S,O) = \sum_{c_i=0}^{k} \sum_{b=0}^{n} \begin{cases} 1 & (O_{c_i,b}, S_{c_i,b}) \in \{(0,0),(1,1)\} \\ \varphi & (O_{c_i,b}, S_{c_i,b}) \in \{(x,x),(z,z)\} \\ -1 & (O_{c_i,b}, S_{c_i,b}) \in \{(1,0),(0,1)\} \\ -\varphi & (O_{c_i,b}, S_{c_i,b}) \in \{(\_,x),(x,\_),(z,\_),(\_,z)\} \end{cases}$$

$$total(S,O) = \sum_{c_i=0}^{k} \sum_{b=0}^{n} \begin{cases} 1 & (O_{c_i,b}, S_{c_i,b}) \in \{(0,0),(1,1),(1,0),(0,1)\} \\ \varphi & (O_{c_i,b}, S_{c_i,b}) \in \{(\_,x),(x,\_),(x,x), \\ & \qquad (z,\_),(\_,z),(z,z)\} \end{cases}$$

For the example shown in Figure 2, the mismatch $x = S_{35,v} \neq O_{35,v} = 0$ subtracts φ from the fitness sum, whereas the match $S_{205,v} = 1 = O_{205,v}$ adds 1 to the fitness sum, with v = overflow_out.

The normalized fitness of the circuit is then defined as:

$$fitness(S,O) = \begin{cases} 0 & sum(S,O) < 0 \\ \frac{sum(S,O)}{total(S,O)} & sum(S,O) \geq 0 \end{cases}$$

This novel approach to calculating normalized fitness is effective at capturing whether or not a candidate design is close to the correct implementation of the circuit, and at guiding the search for a repair.

3.3 Repair Templates

A *repair template* for a defect in code is defined as a pre-identified pattern that can be applied to some aspect of the code to fix the defect. The idea of using templates for APR is well-studied for software [35, 44, 45]. We apply repair templates to aid CirFix in its search for repairs. We propose nine repair templates corresponding to four defect categories for HDL designs. Of the four defect categories we consider, three are suggested in previous work by Sudakrishnan *et al.* [75] that analyzes the bug fix history of four hardware projects written in Verilog and presents several commonly-occurring fixes for HDL descriptions; we propose the remaining defect category based on our experience with defects in hardware designs.

The repair templates in CirFix are presented in Table 1. Incorrect conditionals, sensitivity lists, and assignments correspond to the three most commonly occurring defects in the four hardware projects analyzed in previous work [75, Tab. 2]. Note that our repair templates focus on correct behavior from circuit designs during simulation (cf. rules targeting synthesizability [76]). For an incorrect conditional for a program branch (e.g., the condition for a while-loop or an if-statement), our repair templates can negate the conditional. For an incorrect sensitivity list, recall that in HDL descriptions, a developer can specify blocks of code to execute infinitely often (e.g., line 27, Figure 1a); the execution of such blocks can only be triggered by events described in the block's sensitivity list. Our repair templates for this defect category can modify a block's sensitivity list to change when the block is executed. HDL designs also allow the use of blocking and non-blocking statements for

Table 1: Repair templates in CirFix

Defect Category	Pattern Description
Conditionals	Negate the conditional of a code block (e.g., if-statement, while-loop)
Sensitivity Lists	Trigger an always block on a signal's falling edge Trigger an always block on a signal's rising edge Trigger an always block on any change to a variable within the block Trigger an always block when a signal is level
Assignments	Change a blocking assignment to non-blocking Change a non-blocking assignment to blocking
Numeric	Increment the value of an identifier by 1 Decrement the value of an identifier by 1

assignments. A *blocking* assignment statement (written =) must be executed before any subsequent sequential statements. By contrast, a *non-blocking assignment* (written <=) allows assignments to be made without delaying the procedural flow of a block. Our repair templates for incorrect assignments can change assignments from blocking to non-blocking, and vice versa. Finally, for numeric errors, our repair templates can increment or decrement the values of declared identifiers.

3.4 Repair Operators

CirFix uses two standard repair operators from well-known software repair approaches [39, 47, 63], mutation and crossover, to search the nearby space of circuit designs to produce a repair and to avoid local optima during the process. The input parameter *mutThreshold* (line 11, Algorithm 1) tunes the relative application of mutation and crossover.

As in common software APR approaches (e.g., [39, Sec. III-F]), the mutation operator itself can be characterized into three subtypes: *replace*, *insert*, and *delete*. The mutate function of the CirFix framework generates a random probability value and employs the user-provided replace, insert, and delete thresholds to choose a mutation sub-type. The replace operator picks a random node from the fault localization space and replaces the node with another randomly chosen node from the corresponding fix localization (see Section 3.6) space. The insert operator picks a random node from the fix localization space and inserts it after another randomly picked node within a code block. The delete operator picks a random node from the fault localization and replaces it with an empty node — this operation is equivalent to deleting certain statements from the program variant under consideration.

CirFix uses the standard single-point crossover [62], which starts by picking a *crossover point* for each of the two parents. Edit operations to the right of that point are swapped between the two parents. This results in two children program variants, each carrying some information from both parents. The crossover operator plays a key role in avoiding local optima when searching for high-fitness patches.

3.5 Selection

Automated program repair techniques based on GP use *selection* to choose parent variants from a population based on fitness. *Tournament selection* [56], a selection approach that selects a random pool of t program variants in a population and selects the fittest member of this pool as the parent, has been used widely for software-based APR [39, 42, 63, 81]. CirFix uses tournament selection to select a parent variant to transfer genetic information to the next generation as a child variant. The top e% fittest program variants from the previous generation are automatically chosen to be included in the next generation in a process known as *elitism* [19, 82].

3.6 Fix Localization

Given that fault localization has identified faulty design code to be changed, our *fix localization* provides some guidelines on how to perform the changes. While early works on APR for software chose a node at random for insertion and replacement operations [85], such approaches caused a substantial fraction of mutants to not compile [86]. We use fix localization to restrict the scope of the insert and replace operators to reduce the number of syntactically invalid mutants.

For the insert operator, we propose to only use statements types (e.g., conditional statements, assignments, etc. — see Annex A.6.4 in the IEEE Standard for Verilog [27] for the full BNF definition of statement types) as the sources for insertion code. We further allow such statements to be inserted only into `initial` or `always` blocks, since such statements inserted elsewhere violate the syntax of Verilog [27, Annex A.6.2]. For the replace operator, we design CirFix such that an item in a Verilog module [27, Annex A.1.4] can be replaced either by another item of the same type, or by an item sharing the same immediate parent type (as specified in the formal syntax definition of Verilog [27, Annex A]).

We observe that our fix localization approach reduces the average number of mutants producing compilation errors in our prototype from 35% to 10%. This reduction is comparable to that of fix localization techniques in software (e.g., [39]).

3.7 Repair Minimization

During the search for a repair, CirFix might produce edits to the code that do not contribute to the repair (e.g., repeated assignment statements within an `always` block). Such edits do not increase the fitness of the candidate repair, but they could introduce inefficiencies in the final circuit design or affect the design's readability [66].

CirFix removes such extraneous edits in a postprocessing *minimization* step by finding a subset of the edits in a repair patch from which no further elements can be dropped without causing a reduction in the fitness of the patch. As in APR for software (e.g., [39]), we use the delta debugging algorithm [87] to efficiently (i.e., in polynomial time) compute this *one-minimal* subset of the repair patch. The minimized set of repairs is then converted back into HDL code implementing the hardware design correctly.

4 EXPERIMENTAL SETUP

This section describes the experimental setup for our evaluation of CirFix, including the construction of our new benchmark suite and our choice of experimental parameters.

Table 2: Benchmark hardware projects in our experiments. Project and testbench sizes are measured by source lines of code as reported by the Unix `wc` command.

Project	Description	Project LOC	Testbench LOC
decoder_3_to_8	3-to-8 decoder	25	56
counter	4-bit counter with overflow	56	135
flip_flop	T-flip flop	16	39
fsm_full	Finite state machine	115	66
lshift_reg	8-bit left shift register	30	44
mux_4_1	4-to-1 multiplexer	19	51
i2c	Two-wire, bidirectional serial bus for data exchange between devices	2018	482
sha3	Cryptographic hash function	499	824
tate_pairing	Core for the Tate bilinear pairing algorithm for elliptic curves	2206	983
reed_solomon_ decoder	Core for Reed-Solomon error correction	4366	148
sdram_controller	Synchronous DRAM memory controller	420	95
Total		9770	2923

For our prototype implementation of CirFix, we use the open-source PyVerilog toolkit [79] (version 1.2.1, modified to support numbering for each node type) to parse a Verilog description of a circuit and produce an AST representing the circuit design code. We use Synopsys VCS [78], the primary hardware verification tool used by a majority of the world's top-twenty semi-conductor companies [77], to simulate the code using a manually instrumented testbench to assess functional correctness of the circuit design. Our prototype for CirFix is implemented using Python 3.6.8 and is made publicly available on GitHub (https://github.com/hammad-a/verilog_repair).

4.1 Benchmark Suite for Hardware Defects

For our evaluation of CirFix, we desire a benchmark suite consisting of faulty hardware designs that are indicative of defects in industry, comprise a wide range in terms of project size, and correspond to a variety of components found in real-world designs. To the best of our knowledge, there are no publicly available benchmarks that satisfy our requirements. Additionally, there is limited open source community support for industrial hardware designs, since such designs are often considered Intellectual Property (IP) of the stakeholder companies. As such, we propose to adapt the defect-seeding approach common in software [17, 61, 69] and present a benchmark suite of *defects scenarios* [39, 40] — each consisting of a circuit design, an instrumented testbench for the design, information for correct circuit behavior, and an expert-transplanted defect from real-life experience — to be used for the evaluation of automated repair techniques for hardware.

4.1.1 Selecting Hardware Projects. Every defect scenario includes a base circuit design and a testbench, as introduced in Section 2 (Figure 1). We required circuit designs with an available testbench and that admit simulation using the Synopsys VCS tool without any changes to the design code. This is a common requirement comparable to the benchmarks suites for APR in software [39, Sec. IV-A] [33, Sec. 3.1]. The hardware projects for our benchmark suite are presented in Table 2. For each hardware project, we need an instrumented testbench to record output values for our fitness function. While the instrumentation process is automatable (see Section 3.2), we manually instrument the testbenches for our prototype. Each testbench instrumentation required under 10 lines of Verilog code, took at most 5 minutes of developer time, and did not require any circuit-specific knowledge besides the information already available in the testbench (i.e., identifier names of output wires and registers, and the clock cycle duration).

We choose six projects from undergraduate VLSI courses to be indicative of repairing a small component in hardware design. We augment this by choosing the remaining five projects from OpenCores (a popular website for open-source HDL designs) and GitHub collectively to be indicative of repairing the entirety of a large circuit design. Unlike some previous works that only use toy benchmarks for evaluation (e.g., [12, 74]), our benchmarks include a range of project sizes (in terms of source lines of code), and all projects — including those from courses taught at the undergraduate level — correspond to components found in real-world hardware designs. To satisfy our variety requirement, we include a project from each of the key cores listed on the OpenCores website for OpenCores certified projects (i.e., arithmetic, communication, crypto, error correction, and memory).

4.1.2 Obtaining Information for Correct Circuit Behavior. CirFix requires information about expected behavior for a circuit design to assign fitness values to candidate repairs. In APR for software, guidelines for correct behavior often take the form of passing and failing test cases [46]. More generally, however, such information can be induced from a previous version of the design known to be functional [4, 18, 22, 51, 53, 71] or a combination of data mining and static analyses of the design [15, 23, 25, 72], or manually provided by the human developer [3, 11, 24, 28].

This so-called "oracle problem" [9] remains a challenging issue in general for hardware testing and automated repair: implicit, high-level test oracles (e.g., "the program does not divide by zero") used by APR tools for software do not typically carry over to hardware. Given that circuit designs exhibit parallelism and require synchronization against a clock signal [70], how a circuit design reaches a certain output is often equally important as the actual final output produced. As such, any hardware test oracles need detailed information about the intermediate values from design simulation, and it does not suffice to only use the output values from the simulation as correctness information for an approach like CirFix.

For our benchmark suite, we follow an established approach in APR for software [21, 41] and employ a previously-functioning version of the circuit design to record the expected behavior information for circuits in our benchmark suite. We acknowledge that such a previously-functioning version might not always be available, or the circuit specification may have changed. In that case, a developer can use a partially correct or most up-to-date version of the circuit as a starting point, and manually annotate the missing or incorrect bits based on knowledge of the circuit design. This process is analogous to test suite evolution in software [6]. Ultimately, however, if manual developer effort and previous designs are both unavailable, CirFix cannot be applied to repair defects in a circuit.

While we recognize that the process of manually annotating the correctness information may take longer than manually fixing a single defect, this information is a one-time cost as long as the high-level circuit specification (i.e., I/O wires and registers, expected behavior) does not change. Given the number of bugs that may arise during the development and maintenance of a circuit design, we believe that it would still be more cost effective to invest developer effort in the correctness information, which can then be used by CirFix during inexpensive machine idle time (see discussion in Section 5.1).

4.1.3 Transplanting Hardware Defects. Since actual industrial defects are not made publicly available, we propose an approach based on defect *transplantation* by experts. Previous works have used either randomly-seeded or self-seeded defects for evaluation, potentially admitting bias (e.g., [13]). To combat this, we recruited three hardware experts — two of whom work in industry and one who works in academia, with 19 years of experience with hardware design collectively — to transplant (proprietary or non-public) defects from their real-world experience into otherwise-correct open source implementations of the hardware projects in our benchmark suite. We desire defects in our benchmark suite corresponding to a variety of complexities, both in terms of finding and fixing the defect. As such, we define two defect categories for this process:

- *Category 1:* A Category 1 (i.e., "easy") defect denotes mistakes pertaining to simpler, higher-level aspects of circuit design.
- *Category 2:* A Category 2 (i.e., "hard") defect denotes more intricate errors that usually require more effort to diagnose, understand, and/or fix.

To get the benefits of real-world defects in our benchmark suite, we instructed our recruited experts to transplant and categorize real defects they have previously encountered to the open-source circuits in our benchmark. We also asked our experts for "... variety in how the defects appear and would be fixed, as long as that variety aligns with how often [they] observe these bugs or mistakes in real life". We further required that any transplanted defects should compile successfully and change the externally visible behavior of the circuit with respect to the instrumented testbench, and correspond to approximately the same level of complexity as that of real-world defects.

Table 3 lists the transplanted defects from our experts that met these criteria. In total, our experimental setup includes 32 different defect scenarios spanning across 11 hardware projects, with 19 Category 1 (i.e., "easy") and 13 Category 2 (i.e., "hard") defects. This benchmark suite is 1.5–10× as large as benchmark suites used in the hardware diagnosis literature [12, 13, 29, 65, 74, 75].

4.2 Experimental Parameters

We refer to each execution of CirFix as a *trial*. Each trial is initialized with a distinct random seed for reproducibility of our results, and conducted on a quad-core 3.4GHz machine with hyperthreading and 16GB of memory. We ran 5 independent CirFix trials for each defect scenario, stopping when an acceptable repair was found. Each individual trial was terminated after 8 generations of evolution or 12 hours of wall-clock time (whichever came first).

For the GP parameters, we use population size $popSize = 5000$, repair template threshold $rtThreshold = 0.2$, $mutThreshold = 0.7$. In line with established practices from APR for software [39, 42, 63], we use deletion, insertion, and replacement thresholds of 0.3, 0.3 and 0.4 respectively. For parent selection, we use a tournament size $t = 5$ to increase the selection pressure on candidate variants [57]. For elitism, we propagate the top $e = 5\%$ candidates in each generation to the next generation without any modifications.

For fitness evaluations, we use $\varphi = 2$ as additional weight assigned to bits with values of x or z. This makes incorrect comparisons between ill-defined wires twice as detrimental to the fitness score of a candidate repair as binary bit mismatches. We found that a weight $\varphi = 1$ did not penalize such incorrect comparisons enough (resulting in longer times to find a repair), while $\varphi = 3$ caused too significant a drop in fitness for candidate variants (negatively impacting the exploration of the search space for a repair).

While we leave a comprehensive study of CirFix's parameter sensitivity as future work, we evaluated other values suggested by literature (e.g., smaller population sizes [41, 85]), and found no significant differences in CirFix's performance.

Table 3: Repair results for CirFix. "Cat" indicates the category for the defect, "Repair Time" shows the time for repair (in seconds), and a missing time for repair indicates no repair was found in 5 independent trials. CirFix produced plausible repairs to 21 of the 32 defect scenarios in our benchmark suite, of which 16 were correct upon manual inspection (denoted with a ✓).

Project	Defect Description	Cat	Repair Time (s)
decoder_3_to_8	Two separate numeric errors	1	✓13984.3
	Incorrect assignment	2	—
counter	Incorrect sensitivity list	1	✓19.8
	Incorrect reset	1	✓32239.2
	Incorrect incremental of counter	1	✓27781.3
flip_flop	Incorrect conditional	1	✓7.8
	Branches of if-statement swapped	1	✓923.5
fsm_full	Incorrect case statement	1	—
	Incorrectly blocking assignments	1	4282.2
	Assignment to next state and default in case statement omitted	2	1536.4
	Assignment to next state omitted, incorrect sensitivity list	2	✓37.0
lshift_reg	Incorrect blocking assignment	1	✓14.6
	Incorrect conditional	1	✓33.74
	Incorrect sensitivity list	1	✓7.8
mux_4_1	1 bit instead of 4 bit output	1	—
	Hex instead of binary constants	1	10315.4
	Three separate numeric errors	2	15387.9
i2c	Incorrect sensitivity list	2	✓183
	Incorrect address assignment	2	57.9
	No command acknowledgement	2	✓1560.5
sha3	Off-by-one error in loop	1	✓50.4
	Incorrect bitwise negation	1	—
	Incorrect assignment to wires	2	—
	Skipped buffer overflow check	2	✓50.0
tate_pairing	Incorrect logic for bitshifting	1	—
	Incorrect operator for bitshifting	1	—
	Incorrect instantiation of modules	2	—
reed_solomon_ decoder	Insufficient register size for decimal values	1	—
	Incorrect sensitivity list for reset	2	✓28547.8
sdram_controller	Numeric error in definitions	1	—
	Incorrect case statement	2	—
	Incorrect assignments to registers during synchronous reset	2	✓16607.6

5 EXPERIMENTAL RESULTS

In this section, we present an empirical evaluation CirFix on our benchmark suite of hardware defect scenarios. We address the following research questions:

RQ1. What fraction of defect scenarios can CirFix repair?

RQ2. Does CirFix perform better at repairing Category 1 hardware defects compared to Category 2 defects?

RQ3. How effective is the CirFix fitness function at guiding the search for a repair to a circuit description?

RQ 4. How sensitive is CirFix to the quality of the information for expected behavior?

5.1 RQ1. Repair Rate and Quality for CirFix

Repair Rate. Table 3 presents the repair results for each defect scenario. CirFix produced *plausible* (i.e., testbench-adequate) repairs for 21 of the 32 (65.6%) defects. Of the 11 defects that were not repaired, 4 exhausted resource limits while 7 required edits not supported by CirFix operators and repair templates. While a direct comparison between CirFix and APR for software is not possible, we observe that the repair rate of CirFix comparable to the reported repair rates of well-known software repair approaches, e.g., GenProg (52.4%) [39] and Angelix (34.1%) [55]. When comparing CirFix to a more straightforward search algorithm applying edits at uniform to a circuit design, we found that the brute force algorithm did not scale to the complexity of defects in our benchmark suite and reported no repairs within the 12 hour resource bounds. Though not part of a comprehensive scientific evaluation, when tested on simple single-edit defects (not part of our benchmark suite) in smaller projects from undergraduate courses, the brute-force algorithm still took hours to find repairs that CirFix found in seconds to minutes, highlighting CirFix's efficient pruning of the search space. We leave a full investigation of CirFix against more straightforward search as future work. Note that we can not compare CirFix to other baselines for hardware repair, since at the time of writing, there are no baselines that operate on source code level Verilog descriptions to automatically repair defects; indeed, that is precisely the improvement CirFix brings over the state-of-the-art.

The average wall-clock time for a trial to find a repair was 2.03 hours, of which an average of over 90% was spent on fitness evaluations (i.e., design simulations). Most non-repairs timed out after 12 hours, though defects for some projects with smaller search spaces hit the 8 generation maximum first. These results are in line with previously-reported patterns of behavior for APR for software, supporting our hypothesis that the CirFix algorithm is capable of performing as well on hardware design defects as established APR approaches do on software.

We acknowledge that wall-clock runtime for CirFix on a given defect can be longer than that of an expert human manually fixing the defect. However, CirFix was designed to favor situations in which developer time is significantly more expensive than machine time: it is often more cost-effective to run tools like CirFix using inexpensive machine idle time and then to employ expensive developer time to ensure the repairs are correct before being synthesized [84]. As such, we see CirFix as being cost-effective in terms of reducing the burden on designers.

```
1   1    always @ (posedge clk)
2   2      if (~rst_n)
3   3        begin
4   4        state <= INIT_NOP1;
5   5        command <= CMD_NOP;
6   6        state_cnt <= 4'hf;
7   7        haddr_r <= {HADDR_WIDTH{1'b0}};
8       -
9       -    rd_data_r <= data;
        8  +  state_cnt_next <= 4'd0;
        9  +  rd_data_r <= IDLE;
10  10       busy <= 1'b0;
11  11       end
```

Figure 3: A representative multi-edit repair by CirFix for a defect in the `sdram_controller` benchmark. The original defect, with a missing and an incorrect assignment, is shown in red; the repaired code is shown in green. Edits on lines 8 and 9 correspond to insert and replace operations respectively.

Repair Quality. We follow the approach taken by Long and Rinard [47] and manually analyze the 21 repairs produced by CirFix. We found 5 to be correct and identical to a human repair, another 11 to be correct but different from a human repair, and the final 5 to be correct only with respect to the testbench (i.e., overfitting).[2] While we acknowledge that having a single developer manually examine a patch is not a substitute for a full human study on patch correctness, this analysis adds some confidence that a majority of the plausible repairs from CirFix do not overfit to the testbench (a common problem in APR for software [38, 47, 73]), since we inspect intermediate wire values when assigning fitness scores. Correctness is critical in hardware designs (e.g., since manufactured chips cannot be easily updated once deployed), and we note that our use case does not involve deploying patches directly but instead showing plausible patches to developers to reduce maintenance costs [48, 84].

We observed that 7 out of the 21 minimized repairs were multi-edit repairs, highlighting CirFix's ability to produce repairs to defects that require more than one change to the circuit design. By comparison, common APR approaches for software usually only produce single-edit repairs [21], and only recently have there been works investigating multi-edit repairs [55, 67]. For instance, in a faulty version of the `sdram_controller` benchmark, one of our experts changed assignments to two wires to transplant a Category 2 defect, causing incorrect functionality in the host interface. CirFix assigned this faulty design code a fitness value of 0.818 based on output mismatch. CirFix repaired this defect scenario in 4.6 hours by inserting a new assignment and modifying an existing assignment. The original defect and the repaired code are shown in Figure 3. This is an indicative instance of CirFix repairing Category 2 (i.e., "hard") defects in circuit descriptions with multiple edits to the faulty circuit design.

[2] We focus on correctness of a patch against the specification of the circuit (e.g., ensuring the absence of clock- or reset-domain issues) during our manual inspections. The synthesizability of the design is left to be evaluated by the developer during the validation phase of the hardware design process [80].

CirFix produced plausible repairs to 21 out of 32 (65.6%) defect scenarios in our benchmark suite, of which 16 repairs were fully correct and 5 were correct only with respect to the testbench. The CirFix repair rate is comparable to strong results from APR for software, suggesting that our approach brings the benefits of APR to hardware designs.

5.2 RQ2. Performance for Individual Defect Categories

CirFix found plausible repairs to 12 out of 19 (63.2%) of Category 1 and 9 out of 13 (69.2%) of Category 2 defects. The average number of fitness probes for a trial finding a repair to a Category 1 defect was 9500, taking an average wall-clock time of 2.07 hours to complete. By comparison, the average number of probes for a trial repairing a Category 2 defect was 5000, taking an average wall-clock time of 1.97 hours to complete. We found no statistically significant difference in the average amount of time to find a repair between Category 1 and 2 defects (two-tailed Mann-Whitney U test, $p = 0.373$), suggesting that for defects that CirFix is able to patch, the repair can be produced in about the same time, regardless of the category (and therefore, difficulty) of the defect.

The CirFix repair operators and repair templates were particularly successful at repairing defects of both categories pertaining to incorrect sensitivity lists for `always` blocks, and numeric errors in, or omissions of assignments to wires and registers. On the other hand, CirFix was less successful in defect scenarios where wires or registers are defined incorrectly, or where modules are incorrectly instantiated. For instance, in a Category 1 defect scenario for the `reed_solomon_decoder` project, one of our experts changed the size of a register to 8 bits before assigning a decimal value of 500 to the register. This produces incorrect circuit behavior since 8 bits are not sufficient to store a value of 500. CirFix could not produce a repair to this defect scenario: none of its operators or repair templates are capable of increasing the number of bits allocated to the integer 500. We note that while adding more repair templates can help in such cases, in general, CirFix is able to repair both Category 1 and 2 defects with comparably high success rates.

CirFix performs equally well for Category 1 and Category 2 hardware defects, adding confidence that our approach scales well to a variety of defect types in hardware design.

5.3 RQ3. Quality of Fitness Function

CirFix's high repair rate suggests that our fitness function, coupled with our testbench instrumentation approach, is highly effective at guiding the search for repairs to faulty circuit designs. We observe that for each change to design code that brings a candidate repair closer to a correct repair, our fitness function shows a corresponding increase in the candidate repair's fitness (i.e., our fitness function has a strong *fitness distance correlation*, a trait that makes genetic algorithms thrive [32]). This is best observed in transplanted defects that require multiple edits to the design code to be corrected. For instance, one of our experts transplanted a defect in the `counter` project that required three edits to the design be repaired. The triple-edit repair produced by CirFix for this defect scenario incrementally raised the fitness of the best candidate patch first from 0 to 0.58, then to 0.77, and finally to 1.0 to produce a correct repair. Similar behavior is seen for every other multi-edit repair produced by CirFix, indicating that our fitness function is effective at capturing incremental changes to a circuit design during the search for a repair.

We also observe instances where CirFix produces a repair deemed unfit by our fitness function and instrumented testbench but considered correct by the original, unannotated testbench. We examine one such case in detail, related to the `out_stage` module in the error correction core `reed_solomon_decoder`. This module is responsible for generating output bytes from pipelining input memories. A faulty version of this circuit obtained from one of our experts removed the `reset` wire from the sensitivity list of an `always` block. This caused incorrect resetting of output wires by the circuit. Our fitness function assigns the incorrect design code a non-perfect fitness value of 0.999. The original testbench, however, reports no errors in the incorrect code. The final repair produced by CirFix fixes this defect and passes all checks by the original testbench and our instrumented testbench. This suggests that our fitness function and testbench instrumentation can catch errors beyond the capabilities of the original testbench without adding any additional testing logic.

The CirFix fitness function is highly effective at capturing incremental changes to a circuit's design code to guide the search for a repair, and has the potential to increase testing prowess without any added testing logic to a bench.

5.4 RQ4. Sensitivity to Correctness Information

Since the information for expected circuit behavior is a non-trivial cost for our algorithm, we investigate the quality of the repairs produced by CirFix as a function of the quality of this information. We consider the defects in our benchmark suite repaired under conditions where high quality guidelines for correctness were available, since repairing the remaining defects with lower quality information could be attributed to the randomness associated with a stochastic approach.

As we varied the amount of correctness information (i.e., annotations of expected wire and register values) available from 100% → 50% → 25%, we observed the number of plausible repairs transition from 21 → 20 → 20 and the number of correct repairs go from 16 → 12 → 10. Breaking down the scenario where only 50% of the correctness information was available, we observed that 5 are correct and identical to a human repair, another 7 are correct but different from a human repair, and the final 8 are correct only with respect to the testbench (including a partial repair to a defect requiring multiple edits to be patched). Note that this is a reduction of 25% in the number of correct repairs when the correctness information is reduced by half. Indeed, of these plausible repairs, a total of 10 were identical to repairs produced under conditions when the full expected behavior was available. For the scenario where only a quarter of the expected behavior information was available, we found that 4 are correct and identical to a human repair, another 6 are correct but different from a human repair, and the final 10 are correct only with respect to the testbench (including a partial repairs). This corresponds to a decrease of only 37.5% in

the number of correct patches when the correctness information is reduced by 75%.

Our results indicate that the repair rate, and, more importantly, the quality of the repairs produced by CirFix, is not overly sensitive to the quality of the provided expected behavior information. Furthermore, reducing said behavior information does not increase the manual burden of inspecting produced plausible patches, since CirFix only reports the first plausible patch it finds (cf. program repair for software, where developers may need to evaluate an increasing number of plausible patches as the quality of the test suite is degraded [37, 59]). This analysis gives confidence that even in settings where high quality information for correct circuit behavior might not be available, high marginal benefit and reduction in maintenance costs are still obtainable from CirFix.

> CirFix is not overly sensitive to the quality of the expected circuit behavior information, yielding high repair rates and quality even under settings when low quality correctness information is used as input to the algorithm.

6 LIMITATIONS AND THREATS TO VALIDITY

Our results in Section 5 suggest that CirFix is highly effective at automatically repairing defects in HDL descriptions. That said, there are several limitations to our approach and threats to the validity of our results that we describe in this section.

Timing bugs. Faults in HDL descriptions stemming from timing flow issues and incorrect circuit behavior with respect to the clock signal often go undetected by a traditional testbench, requiring instead complicated analyses of waveforms from the simulation. Such timing bugs are therefore not in scope of our approach that heavily relies on testbenches to assess functional correctness of designs. We note that while such bugs are complex to debug, they represent only a subset of hardware defects in industry, and a non-trivial amount of defects in hardware correspond to functional correctness [16].

Threats to Validity. The parameters for the prototype implementation of CirFix are chosen based on empirical performance and may not be optimal. We do note, however, that the repair operators, fault and fix localization approaches, and representation choice for repairs matter more than the actual values of the GP parameters for APR [7].

Our benchmark defects may not be indicative of defects in real-world hardware projects, posing a potential threat to external validity. To mitigate this threat, we evaluated CirFix on a variety of hardware projects taken from different sources, and had expert hardware designers transplant defects from their real-life experience with HDL designs covering a variety of defect types (see Section 4.1.3).

7 RELATED WORK

Automatic Error Diagnosis and Correction in Hardware Designs. While a significant amount of work has been done in automatic error diagnosis of hardware designs, the correction of such errors automatically has not been well-explored to the best of our knowledge. Techniques in the works of Jiang *et al.* [29] and Ran *et al.* [65] employ software analysis approaches to identify statements in design code responsible for defects, but suffer from high false positive rates. Bloem and Wotawa [12] use formal analysis of circuit descriptions to identify defects, but their approach requires formal specifications for large real-world designs that are not always available. Staber *et al.* [74] use state-transition analysis to diagnose and correct hardware designs automatically, but their techniques similarly do not scale to real-world circuits with large state spaces. Our approach, by contrast, is more scalable to large, real-world hardware descriptions. Chang *et al.* [13] explicitly insert multiplexers to automatically diagnose faults in hardware designs and suggest repairs; Madre *et al.* [49] use Boolean equation solving to diagnose and rectify gate-level design errors. By contrast, our technique applies to both behavioral (higher level) and RTL aspects of a circuit design.

Automated Program Repair for Software. In the realm of software, significant research effort has been devoted to repairing bugs automatically over the last decade [21, 46, 58]. Automated program repair usually takes as input source code with a deterministic bug and a test suite with at least one failing test that reveals the bug, and aims to automatically generate fixes to the buggy code. Test suite based repair, where test cases are used to guide the search for a patch, can be further divided into generate-and-validate and semantics-driven approaches. Generate-and-validate techniques produce candidate patches for the buggy code and evaluate them against the test suite to check if all tests pass [2, 39, 63, 64]. Semantics-driven approaches first extract constraints on a program based on test suite execution and then use these constraints to synthesize a patch [52, 54, 55, 61]. While software approaches to APR make use of test suites to evaluate candidate repairs, CirFix uses instrumented hardware testbenches to make visible the internal and external behavior of a simulated circuit for fitness evaluation. Additionally, APR for software usually uses spectrum-based fault localization to implicate faulty code, whereas CirFix uses our novel fault localization approach supporting the analysis of parallel hardware descriptions.

8 CONCLUSION

This paper presents CirFix, a framework for automatically repairing defects in hardware designs implemented in languages like Verilog. CirFix makes use of readily-available artifacts included in the hardware design process (e.g., testbenches) to diagnose and repair defects in both behavioral and RTL designs in the circuit description. These repairs can then be shown to developers for validation before the synthesis phase, reducing maintenance costs. The testbench-based evaluation and the parallel structure of hardware designs pose challenges that render traditional APR approaches from software inapplicable to the hardware domain. We present two key insights to bridge this gap. First, we propose a method to instrument hardware testbenches to admit a circuit's behavior to guide the search for repairs. We present a novel fitness function tailored that performs a bit-level comparison of the made-visible output wire values against expected behavior to assess functional correctness of candidate repairs. Second, we present a novel fault localization approach based on a fixed point analysis of assignments made to registers and output wires to implicate statements for defects. Our systematic evaluation of CirFix presents a new

benchmark suite of 32 defect scenarios transplanted by three hardware experts across 11 different Verilog projects. CirFix produces plausible repairs for 21 out of 32 and fully correct repairs for 16 out of 32 of the Verilog defects within reasonable resource bounds.

ACKNOWLEDGMENTS

We gratefully acknowledge the partial support of the NSF (CCF 1908633, CCF 1763674) and a Google Faculty Research Award. Toyota Research Institute ("TRI") provided funds to partially assist the authors with their research, but this article solely reflects the opinions and conclusions of its authors and not TRI or any other Toyota entity.

A ARTIFACT APPENDIX

A.1 Abstract

We provide the public repository for CirFix, both on Zenodo and GitHub. The artifact includes instructions for installing and running CirFix, as well as scripts and instructions used to reproduce core results from our paper.

A.2 Artifact Check-List (Meta-Information)

- **Program:** python3.6.8, pyverilog-1.2.1, iverilog, VCS
- **Run-time environment:** Red Hat Enterprise Linux 7.9
- **Hardware:** Intel quad-core 3.4GHz machine with hyperthreading and 16GB of memory
- **Output:** Repair patchlist (i.e., sequence of edits to source code) fixing a defect, if a repair is found
- **Experiments:** Running CirFix on the defects in our benchmark suite, runtime analysis of CirFix
- **How much disk space required (approximately)?:** 5GB
- **How much time is needed to prepare workflow (approximately)?:** <1 hour
- **How much time is needed to complete experiments (approximately)?:** 15-20 hours (longer experiments can be run concurrently and overnight)
- **Publicly available?:** Yes (GitHub repository: https://github.com/hammad-a/verilog_repair)
- **Code licenses (if publicly available)?:** MIT License
- **Archived (provide DOI)?:** Available at Zenodo: https://doi.org/10.5281/zenodo.5846419

A.3 Description

The artifact contains all of CirFix's source code as well as the instructions to install and run CirFix and its dependencies. The README.md files at the root of the repository and the /prototype and /pyverilog_changes directories contain all of the instructions used in the Artifact Evaluation process.

A.3.1 How to Access the Artifact. CirFix is available at our public GitHub repository as well as an archive on Zenodo (see Appendix A.2).

A.3.2 Hardware Dependencies. The artifact does not have any explicit dependencies, though older, slower hardware might take slightly longer to reproduce our results. For our experiments, we used an Intel quad-core 3.4GHz machine with hyperthreading and 16GB of memory.

A.3.3 Software Dependencies. CirFix requires Python 3.6.8, PyVerilog version 1.2.1, and Icarus Verilog. It also requires Synopsys VCS simulator (commercial license) to simulate Verilog designs.

A.4 Installation

This section assumes that users already have access to the Synopsys VCS tool. Note that alternative Verilog simulation tools may be used, but would likely require modifications to the scripts to support the API for the simulation tool.

Users first need to install Python 3.6.8 and all external Python dependencies (listed under the README.md file at the root of the repository). Some source files for PyVerilog need to be changed to support CirFix; instructions to do so can be found at /pyverilog_changes/README.md. Users then need to configure CirFix to run on a defect by editing the configuration file (located at /prototype/repair.conf). This involves setting the source file, testbench, correctness information, and evaluation script paths. Users also need to configure the CirFix GP parameters if necessary (we include our default values in the configuration file). The detailed instructions for this process are included in the README file located at /prototype/README.md.

A.5 Experiment Workflow

After all dependencies have been installed and the configuration file set, users may run the outer CirFix script (/prototype/repair.py) to start a CirFix run using the terminal command `python3 repair.py`. The script invokes calls to PyVerilog to parse the Verilog source code into a program AST, which is then manipulated to edit the source code. For every change to the AST, CirFix re-generates the Verilog source code before passing it on to the Synopsys VCS simulator, which in turn uses the new code and the provided testbench to generate the circuit output on given input stimuli. The produced circuit output is then passed back to CirFix and compared against developer provided circuit behavior information to assess the correctness of the produced circuit design. CirFix terminates when it finds a design producing output that matches expected behavior. Users may pass the `log=true` flag to store detailed logs in the /prototype/repair_logs directory.

A.6 Evaluation and Expected Results

The artifact provides instructions for reproducing the main results from CirFix's evaluation (Table 3). Every CirFix execution (or trial) that finds a repair to a bug ends with the minimized repair patchlist (i.e., a sequence to edits to the source code that ultimately repair the defect). This repair patchlist can be verified against our reported results (Table 3; raw data included in /prototype/experiments_results.xlsx), along with the time to find the repair. We also provide instructions on how to use this repair patchlist to produce Verilog source code for inspection in the file /prototype/README.md.

A.7 Notes

We hope to maintain CirFix as an open-source tool. Any issues that are found with the available artifact or any questions that arise can be submitted as GitHib issues on our repository or communicated via email.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792.

[2] Thomas Ackling, Bradley Alexander, and Ian Grunert. 2011. Evolving patches for software repair. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. 1427–1434.

[3] Sheeva Afshan, Phil McMinn, and Mark Stevenson. 2013. Evolving Readable String Test Inputs Using a Natural Language Model to Reduce Human Oracle Cost. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 352–361. https://doi.org/10.1109/ICST.2013.11

[4] KK Aggarwal, Yogesh Singh, Arvinder Kaur, and OP Sangwan. 2004. A neural net based approach to test oracle. *ACM SIGSOFT Software Engineering Notes* 29, 3 (2004), 1–6.

[5] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. 1995. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. IEEE, 143–151.

[6] Nada Alsolami, Qasem Obeidat, and Mamdouh Alenezi. 2019. Empirical analysis of object-oriented software test suite evolution. *International Journal of Advanced Computer Science and Applications* 10, 11 (2019).

[7] Andrea Arcuri and Gordon Fraser. 2011. On parameter tuning in search based software engineering. In *International Symposium on Search Based Software Engineering*. Springer, 33–47.

[8] Desire Athow. 2014. Pentium fdiv: The processor bug that shook the world. https://www.techradar.com/news/computing-components/processors/pentium-fdiv-the-processor-bug-that-shook-the-world-1270773

[9] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[10] Lionel Bening and Harry Foster. 2001. RTL Formal Verification. *Principles of Verifiable RTL Design: A functional coding style supporting verification processes in Verilog* (2001), 103–129.

[11] Robert Binder. 2000. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional.

[12] Roderick Bloem and Franz Wotawa. 2002. Verification and fault localization for VHDL programs. *Journal of the Telematics Engineering Society (TIV)* 2 (2002), 30–33.

[13] Kai-hui Chang, Ilya Wagner, Valeria Bertacco, and Igor L Markov. 2007. Automatic error diagnosis and correction for RTL designs. In *2007 IEEE International High Level Design Validation and Test Workshop*. IEEE, 65–72.

[14] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*. 559–572.

[15] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. 2016. Supporting oracle construction via static analysis. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 178–189.

[16] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2019. Hardfails: Insights into software-exploitable hardware bugs. In *USENIX Security Symposium*. 213–230.

[17] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10, 4 (2005), 405–435.

[18] Robert Feldt. 1998. Generating diverse software versions with genetic programming: an experimental study. *IEE Proceedings-Software* 145, 6 (1998), 228–236.

[19] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. 947–954.

[20] Harry Foster. 2008. Assertion-based verification: Industry myths to realities (invited tutorial). In *International Conference on Computer Aided Verification*. Springer, 5–10.

[21] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2017), 34–67.

[22] Farshad Gholami, Niousha Attar, Hassan Haghighi, Mojtaba Vahidi Asl, Meysam Valueian, and Saina Mohamadyari. 2018. A classifier-based test oracle for embedded software. In *2018 Real-Time and Embedded Systems and Technologies (RTEST)*. 104–111. https://doi.org/10.1109/RTEST.2018.8397165

[23] Mohamed Hanafy, Hazem Said, and Ayman M. Wahba. 2015. New methodology for digital design properties extraction from simulation traces. In *2015 Tenth International Conference on Computer Engineering Systems (ICCES)*. 91–98. https://doi.org/10.1109/ICCES.2015.7393026

[24] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. 2010. Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle Cost Problem. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. 182–191. https://doi.org/10.1109/ICSTW.2010.31

[25] Samuel Hertz, David Sheridan, and Shobha Vasudevan. 2013. Mining Hardware Assertions With Guidance From Static Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 6 (2013), 952–965. https://doi.org/10.1109/TCAD.2013.2241176

[26] Yu Huang, Hammad Ahmad, Stephanie Forrest, and Westley Weimer. 2021. Applying Automated Program Repair to Dataflow Programming Languages. In *GI @ ICSE 2021*, Justyna Petke, Bobby R. Bruce, Yu Huang, Aymeric Blot, Westley Weimer, and W. B. Langdon (Eds.). IEEE, internet.

[27] IEEE. 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), 1–590. https://doi.org/10.1109/IEEESTD.2006.99495

[28] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 247–258.

[29] Tai-Ying Jiang, C-NJ Liu, and Jing Ya Jou. 2005. Estimating likelihood of correctness for error candidates to assist debugging faulty HDL designs. In *2005 IEEE International Symposium on Circuits and Systems*. IEEE, 5682–5685.

[30] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.

[31] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 467–477.

[32] Terry Jones and Stephanie Forrest. 1995. Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms. In *ICGA*, Vol. 95. 184–192.

[33] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440.

[34] Robert Keim. 2020. What is a Hardware Description Language (HDL)? Retrieved Jan 11, 2021 from https://www.allaboutcircuits.com/technical-articles/what-is-a-hardware-description-language-hdl/.

[35] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 802–811.

[36] John R Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press.

[37] Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On Reliability of Patch Correctness Assessment. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 524–535. https://doi.org/10.1109/ICSE.2019.00064

[38] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *Empirical Software Engineering* 23, 5 (2018), 3007–3033.

[39] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 3–13.

[40] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Software quality journal* 21, 3 (2013), 421–443.

[41] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[42] Claire Le Goues, Westley Weimer, and Stephanie Forrest. 2012. Representations and operators for improving evolutionary software repair. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. 959–966.

[43] Robert Lemos. 1997. Intel releases fix FOR F00F bug. https://www.zdnet.com/article/intel-releases-fix-for-f00f-bug/

[44] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 31–42.

[45] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 456–467.

[46] Yuzhen Liu, Long Zhang, and Zhenyu Zhang. 2018. A Survey of Test Based Automatic Program Repair. *JSW* 13, 8 (2018), 437–452.

[47] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 298–312.

[48] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 75–87. https://doi.org/10.1145/3395363.3397351

[49] J. C. Madre, O. Coudert, and J. P. Billon. 1989. Automating the diagnosis and the rectification of design errors with PRIAM. In *1989 IEEE International Conference*

on Computer-Aided Design. Digest of Technical Papers. 30–33. https://doi.org/10.1109/ICCAD.1989.76898

[50] M Morris Mano and Michael Ciletti. 2013. *Digital design: with an introduction to the Verilog HDL*. Pearson.

[51] L. I. Manolache and Derrick G. Kourie. 2001. Software testing using model programs. *Software: Practice and Experience* 31, 13 (2001), 1211–1236.

[52] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA*. https://doi.org/10.1145/2931037.2948705

[53] Phil McMinn. 2009. Search-based failure discovery using testability transformations to generate pseudo-oracles. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. 1689–1696.

[54] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 448–458.

[55] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. 691–701.

[56] Brad L Miller and David E Goldberg. 1996. Genetic algorithms, selection schemes, and the varying effects of noise. *Evolutionary computation* 4, 2 (1996), 113–131.

[57] Brad L Miller, David E Goldberg, et al. 1995. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems* 9, 3 (1995), 193–212.

[58] Martin Monperrus. 2018. *The Living Review on Automated Program Repair*. Technical Report hal-01956501. HAL/archives-ouvertes.fr.

[59] Manish Motwani. 2021. High-Quality Automated Program Repair. arXiv:2104.07851 [cs.SE]

[60] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 1–32.

[61] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[62] Riccardo Poli and William B Langdon. 1998. Genetic programming with one-point crossover. In *Soft Computing in Engineering Design and Manufacturing*. Springer, 180–189.

[63] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*. 254–265.

[64] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 24–36.

[65] Jiann-Chyi Ran, Yi-Yuan Chang, and Chia-Hung Lin. 2003. An efficient mechanism for debugging RTL description. In *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, 2003. Proceedings.* IEEE, 370–373.

[66] Simone Romano, Christopher Vendome, Giuseppe Scanniello, and Denys Poshyvanyk. 2018. A multi-study investigation into dead code. *IEEE Transactions on Software Engineering* (2018).

[67] Seemanta Saha et al. 2019. Harnessing evolution for multi-hunk program repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 13–24.

[68] Frank Schirrmeister, Michael McNamara, Larry Melling, and Neeti Bhatnagar. 2012. Debugging at the hardware/software interface. https://www.embedded-computing.com/embedded-computing-design/debugging-at-the-hardware-software-interface

[69] Eric Schulte, Zachary P Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014. Software mutational robustness. *Genetic Programming and Evolvable Machines* 15, 3 (2014), 281–312.

[70] Charles L Seitz, C Mead, and L Conway. 1980. System timing. *Introduction to VLSI systems* (1980), 218–262.

[71] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. 2011. An automated framework for software test oracle. *Information and Software Technology* 53, 7 (2011), 774–788. https://doi.org/10.1016/j.infsof.2011.02.006

[72] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, and Siti Zaiton Mohd-Hashim. 2009. A Comparative Study on Automated Software Test Oracle Methods. In *2009 Fourth International Conference on Software Engineering Advances*. 140–145. https://doi.org/10.1109/ICSEA.2009.29

[73] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 532–543.

[74] Stefan Staber, Barbara Jobstmann, and Roderick Bloem. 2005. Finding and fixing faults. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 35–49.

[75] Sangeetha Sudakrishnan, Janaki Madhavan, E James Whitehead Jr, and Jose Renau. 2008. Understanding bug fix patterns in verilog. In *Proceedings of the 2008 international working conference on Mining software repositories*. 39–42.

[76] Stuart Sutherland. 2017. *RTL Modeling with SystemVerilog for Simulation and Synthesis Using SystemVerilog for ASIC and FPGA Design*. Sutherland HDL, Incorporated.

[77] Synopsys. 2020. VCS Functional Verification Solution. https://www.synopsys.com/verification/simulation/vcs.html

[78] VCS Synopsys. 2004. Verilog Simulator. *Avaliable HTTP: http://www. synopsys. com/products/simulation/simulation. html* (2004).

[79] Shinya Takamaeda-Yamazaki. 2015. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *International Symposium on Applied Reconfigurable Computing*. Springer, 451–460.

[80] Vaibbhav Taraate. 2016. *Digital logic design using verilog: coding and RTL synthesis*. Springer.

[81] Christopher S Timperley. 2017. *Advanced techniques for search-based program repair*. Ph. D. Dissertation. University of York.

[82] JA Vasconcelos, Jaime Arturo Ramirez, RHC Takahashi, and RR Saldanha. 2001. Improvements in genetic algorithms. *IEEE Transactions on magnetics* 37, 5 (2001), 3414–3417.

[83] Jayce Wagner. 2018. Intel Could Make Billions Off of Meltdown & Spectre. https://www.digitaltrends.com/computing/intel-could-make-billions-off-meltdown-spectre/

[84] Westley Weimer. 2006. Patches as Better Bug Reports. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering* (Portland, Oregon, USA) *(GPCE '06)*. Association for Computing Machinery, New York, NY, USA, 181–190. https://doi.org/10.1145/1173706.1173734

[85] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. 2010. Automatic program repair with evolutionary computation. *Commun. ACM* 53, 5 (2010), 109–116.

[86] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 364–374.

[87] Andreas Zeller. 2001. Automated debugging: Are we close. *Computer* 11 (2001), 26–31.