CirFix: Automated Hardware Repair and its Real-World Applications

Priscila Santiesteban, Yu Huang, Westley Weimer, Hammad Ahmad

Abstract—This article 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. Additionally, we evaluate CirFix's fault localization independently through a human study (n=41), and find that the approach may be a beneficial debugging aid for complex multi-line hardware defects.

Index Terms—Circuit Designs, Automated Repair, Empirical Study, User Study

1 Introduction

Recent increases in the complexity of hardware designs have challenged the ability of developers to find and repair defects in circuit descriptions [1]. 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 [2]. Indeed, recent functional and security vulnerabilities due to defects at the hardware design level have led to expensive consequences [3], [4], [5]. 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 [6], [7]. Other approaches for automatic error diagnosis and correction require formal specifications to conduct design verification [8], which usually do not scale to large designs.

Furthermore, previous work does not operate on behavioral-level descriptions of hardware circuits [9], [10]. On the other hand, in the realm of software, significant research effort focuses on repairing bugs automatically [11], [12], [13]. *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. Interest in applying software APR methods to hardware has been seen in the literature. Some methods for localizing hardware errors focus on applying a model-based diagnosis paradigm and making use of structure and behavior for software debugging [14], [15], [16], [17].

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) HDL designs are inherently parallel and often include non-sequential statements, since separate portions of hardware can operate simultaneously. While some conventional languages, such as Javascript, have support for parallelism, APR typically focuses on software written in languages such as C and Java that are generally based around a serial execution model. (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* [18], 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 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 a more parallel structure [19], dataflow-based fault localization (e.g., [20]) 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 [21]. 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. Our novel fault localization utilizes the simulations to assign blame to incorrect wires and registers.[1] 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 present a benchmark suite of 32 *defect scenarios* [22] 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.

Furthermore, we evaluate the usability of our novel fault localization algorithm independent of the automated repair context through a human study in which $n = 41$ humans assess its quality and usefulness. We find a statistically-significant preference ($p = 0.003$) for CirFix fault localization as a debugging aid in fixing multi-line hardware defects, primarily in student applications ($p = 0.01$).

The main contributions of this paper are:

- CirFix, a repair algorithm for hardware designs.
- 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 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.
- A human study using CirFix's fault localization algorithm as a debugging aid on real-world and student applications. We observe statistically significant preference using the support for multi-line defects ($p = .003$) in student applications ($p = 0.01$).

Additions relative to prior paper. This article extends our ASPLOS 2022 paper [23], but also includes (1) a new human study of the proposed fault localization algorithm for hardware designs, (2) an independent assessment of correctness of the produced CirFix patches from an expert team [24], [25], [26], [27], [28], (3) an investigation of fault localization sensitivity and, (4) a discussion of the degree to which CirFix interacts with the synthesizability or timing of a design. In Section 4.3 we introduce a human study to investigate the incremental benefit of our fault localization in various Verilog debugging scenarios. In Section 5.1 we report an assessment of CirFix patches along an orthogonal evaluation criterion from an independent expert APR team. In Section 6 we analyze the results in terms of objective performance and subjective judgements. In Section 5.1, we evaluate CirFix's repair performance by conducting a targeted experiment that controls the quality of initially-provided information. Lastly, we discuss synthesizability and timing of repairs in Section 7.

1. In HDLs like Verilog, wire elements are used to connect input and output ports of a module instantiation, while registers stores values.

```
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

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

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 [29].

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. Note that there also exists a notion of asynchronous designs where the state of the system can change in response to changing inputs. However, given the increased complexity associated with asynchronous designs, most hardware designs tend to be synchronous in nature [30].

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 if-statement that resets the wires (line 30, Figure 1a) to evaluate the true branch, after which the counter is reset. A correct design should also reset the overflow bit: at this point, the expected behavior 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.

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, and thus supports HDL programming constructs such as sequential and parallel code, variable reassignment, and synchronized code blocks. Our prototype would require modifications to generalize to other hardware description languages (e.g., adding support for AST parsing or different simulation environments). The pseudocode 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, scores 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) [31], an iterative stochastic search technique, to synthesize candidate repairs to faulty HDL programs. The

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, $popSize$, $maxGens$, $rtThreshold$, $mutThreshold$.
Output: Repaired circuit description.

1: $pop \leftarrow$ seed_pop$(C, popSize)$
2: **repeat**
3: $\quad childPop \leftarrow \emptyset$
4: $\quad$ **while** $|childPop| \leq popSize$ and $\forall\ candidate \in childPop.\ f(candidate, TB, O) < 1.0$ **do**
5: $\quad\quad parent \leftarrow$ tournament_selection(pop, f)
6: $\quad\quad fl\_set \leftarrow$ fault_loc$(parent)$
7: $\quad\quad$ **if** probability$() \leq rtThreshold$ **then**
8: $\quad\quad\quad child \leftarrow$ apply_fix_pattern$(parent, fl_set)$
9: $\quad\quad\quad childPop \leftarrow childPop \cup \{child\}$
10: $\quad\quad$ **else** ▷ Repair operators
11: $\quad\quad\quad$ **if** probability$() \leq mutThreshold$ **then**
12: $\quad\quad\quad\quad child \leftarrow$ mutate$(parent, fl_set)$
13: $\quad\quad\quad\quad childPop \leftarrow childPop \cup \{child\}$
14: $\quad\quad\quad$ **else**
15: $\quad\quad\quad\quad parent2 \leftarrow$ tournament_selection(pop, f)
16: $\quad\quad\quad\quad \{c1, c2\} \leftarrow$ crossover$(parent, parent2)$
17: $\quad\quad\quad\quad childPop \leftarrow childPon \cup \{c1, c2\}$
18: **until** resources exhausted or $\exists\ candidate \in childPop.\ f(candidate, TB, O) = 1.0$
19: **return** minimize$(candidate, TB, O)$

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,[2] and the input parameters. The algorithm starts with the original source code and maintains a population of program variants, each stored as a *repair patch* [32] 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 3.3) 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 3.4). Our *fix localization* identifies code to be inserted or replaced as part of mutation operations (see Section 3.5). 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* [33] a candidate repair to remove extraneous operations not needed to obtain correct circuit

2. CirFix does not require perfect information for expected behavior for every timestep: the developer can choose to only provide information at certain intervals. See prior work RQ4 [23] for an evaluation of the trade-off between the level of detail of expected output and repair success.

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: $FL, mismatch \leftarrow \emptyset, \emptyset$
2: $mismatch' \leftarrow$ get_output_mismatch(O, S) ▷ Section 3.2
3: **while** $mismatch \neq mismatch'$ **do**
4: $\quad mismatch \leftarrow mismatch \cup mismatch'$
5: $\quad$ **for** $node$ in ast **do**
6: $\quad\quad$ **if** implicated$(node, mismatch)$ **then**
7: $\quad\quad\quad FL \leftarrow FL \cup \{node.id\}$
8: $\quad\quad\quad$ **for** each $child$ of $node$ **do**
9: $\quad\quad\quad\quad FL \leftarrow FL \cup \{child.id\}$
10: $\quad\quad\quad\quad$ **if** type$(child) =$ Identifier and $child \notin mismatch$ **then**
11: $\quad\quad\quad\quad\quad mismatch' \leftarrow mismatch' \cup \{child\}$
12: **return** FL

output (line 19 in Algorithm 1; see Section 3.6). 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 [34]) for validation before the design is ultimately synthesized, reducing maintenance costs [35], [36].

3.1 Fault Localization

Fault localization is critical to the success and efficiency of APR [37]. Traditional APR for software often relies on spectrum-based fault localization [38] 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 [19].

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 [39]. 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 [40], [41], [42], 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 simulated and expected outputs). It then compares the simulation output against the expected behavior to produce a set of *identifiers* (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[3]. 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) becomes implicated by (Impl-Ctrl), bringing in the new identifier `counter_out` to the mismatch set by (Add-Child). The process is repeated until no new identifiers are added to the mismatch set.

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. Note that while we demonstrate the scalabaility of our approach on a variety of hardware designs of different sizes (see Table 2), our approach may require additional developer effort to generalize to very complex designs (e.g., a microprocessor) with millions of wires, gates, and registers. We discuss this limitation in Section 8.

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* [43] (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 mentioned in Section 1, traditional APR for software uses test-case based evaluation strategies to assess candidate repairs. Hardware designs, by contrast, use testbenches to verify functional correctness (see Section 1 for details on the difference between hardware and software evaluations). 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*.

3. In a focused investigation of our our three largest benchmarks, both control flow complexity and also the number of wires/registers were found to contribute equally (40–50% each) to the final fault localization size, and thus the scalability of our algorithm.

Many 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 [44]. 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, $S_{c_i,b} = S(c_i(b))$. 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:

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

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

The normalized fitness of the circuit is then defined as:

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

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

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 & Repair Operators

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 [45], [46], [47]. 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.* [39] 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 [39, Tab. 2]. Note that our repair templates focus on correct behavior from circuit designs during simulation (cf. rules targeting synthesizability [48]). 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.

CirFix uses two standard repair operators from well-known software repair approaches [22], [49], [50], mutation and crossover, to search the nearby space of circuit designs to produce a repair and to avoid local optima. The input parameter $mutThreshold$ (line 11, Algorithm 1) tunes the relative application of mutation and crossover.

As in common software APR approaches (e.g., [22, 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 3.5) 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 [51], which picks 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.

3.4 Selection

Automated program repair techniques based on GP use *selection* to choose parent variants from a population based on fitness. *Tournament selection* [52], 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 [22], [49], [53], [54]. 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 included in the next generation, a process known as *elitism* [55], [56].

3.5 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. We use fix localization to restrict the scope of the insert and replace operators to reduce the number of syntactically-invalid mutants (cf. [57]).

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 [58] 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 [58, Annex A.6.2]. For the replace operator, we design CirFix such that an item in a Verilog module [58, 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 [58, Annex A]). We return to this decision in Section 7.

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., [22]).

3.6 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 [59].

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., [22]), we use the delta debugging algorithm [33] 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, our choice of experimental parameters, and our human study on evaluating the usability of CirFix's novel fault localization.

For our prototype implementation of CirFix, we use the open-source PyVerilog toolkit [60] (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 [61], the primary hardware verification tool used by a majority of the world's top-twenty semi-conductor companies [62], 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 [63], [64], [65] and present a benchmark suite of *defects scenarios* [22], [37] — 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 [22, Sec. IV-A] [66, 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 beyond that available in the testbench (i.e., identifier names of output wires and registers, and the clock cycle duration).

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

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., [8], [67]), 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 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 [13]. More generally, however, such information can be induced from a previous version of the design known to be functional [68], [69], [70], [71], [72], [73] or a combination of data mining and static analyses of the design [74], [75], [76], [77], or manually provided by the human developer [78], [79], [80], [81].

This so-called "oracle problem" [82] 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 [83], 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 [11], [84] 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 [85]. 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., [9]). 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
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 by the authors (denoted with a ✓) and 14 were deemed correct along a different criteria by an independent expert team (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 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

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 [6], [7], [8], [9], [39], [67].

4.2 Algorithm 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 [22], [49], [53], 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 [86]. For elitism, we propagate the top $e = 5\%$ of each generation to the next 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).

We evaluated other values suggested by literature (e.g., smaller population sizes [84], [87]), and found no significant differences in CirFix's performance.

4.3 Human Study Protocol

We also investigate the usability of our novel fault localization algorithm (see Section 3.1), independent of the automated repair context. We asked humans (i.e., hardware engineers), rather than CirFix, to assess the quality and usefulness of the fault localization algorithm. To investigate the incremental benefit of our fault localization, we consider three scenarios: the full output of the algorithm (see Section 3.1), only initially implicated statements of the algorithm (no transitive information, only line 1 of Algorithm 2), and no fault localization annotations.

Participant Recruitment. Under UM IRB-HUM00199335, we recruited a combination of undergraduate and graduate computer science students ($n = 41$). One student reported having less than a month of experience, ten students reported having 1 to 4 months experience, seven students reported having 4 months to 1 year of experience, nine reported having 1 to 2 years of experience, and the remaining six reported having 2 or more years of experience. We drew students from five undergraduate courses, a graduate course, and a computer engineering lab mailing list at the University of Michigan. At the beginning of the survey, participants' background in Verilog was collected (e.g., any courses they have taken). Participant data was anonymized, but they could optionally request a $25 USD gift card as compensation. We collected 30

Debugging Scenarios. We sampled (uniformly at random) 10 defect scenarios each from student and OpenCores projects, with roughly equal numbers of Category 1 and 2 defects. To favor readability and comprehension within a time-constrained human study (e.g., [88], [89]), we filtered out defects that resulted in more than 100 lines of code implicated by fault localization. This resulted in 12 snippets from the programs in Table 2: eight from student projects and four from OpenCore projects. Each debugging scenario included information on the parent hardware design and documentation on the desired properties and output.

Debugging Task. Each participant was sequentially presented with 6 distinct randomly-chosen debugging scenarios. Each scenario was paired with a debugging hint: textual highlighting of implicated code, as shown in Figure 2.

Participants were asked to: (1) identify faulty lines in the snippet, (2) indicate which lines they would alter to fix the defect, (3) propose how they would alter the lines to fix the defect if they could patch it. If the snippet version presented to the participant contained fault localization hints, the participant also rated the usefulness and accuracy of those hints on a 1–5 scale.

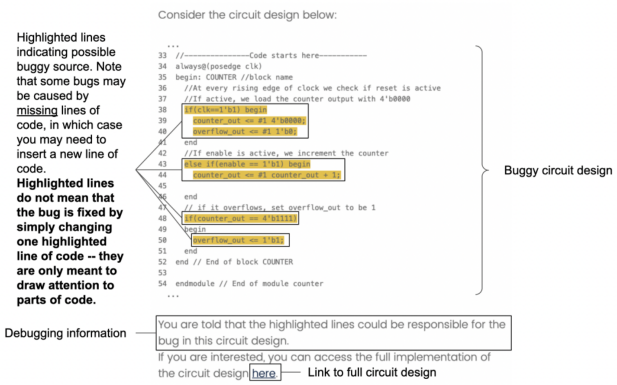

Fig. 2. Example of defect scenario presented to participants.

5 CirFix Repair Evaluation

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

RQ1. What fraction of defect scenarios can CirFix repair, and how sensitive is our fault localization approach?

RQ2. How effective is the CirFix fitness function at guiding the search for a repair to a circuit description?

In prior work [23], we addressed two additional research questions: (1) what is the performance of CirFix on repairing two different types of defects varying in difficulty and (2) how sensitive is CirFix to the quality of the information for expected behavior. We found CirFix to repair both Category 1 and 2 defects with comparably high success rates, and found CirFix to not be 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.

5.1 RQ1. Repair Rate, Quality, and Sensitivity 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%) [22], Angelix (34.1%) [90], and TBar (53.1%) [46]. 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 [35]. As such, we see CirFix as being cost-effective in terms of reducing the burden on designers.

Repair Quality. We follow the approach taken by Long and Rinard [50] for patch assessment since it follows best practices in the APR literature [43], [91]. We manually analyze the 21 repairs produced by CirFix. We found 16 of the generated repairs to exhibit correct behavior, with the final 5 to be correct only with respect to the testbench (i.e., overfitting).[4] While room for improvement remains, software industrial deployments with similar rates have proved useful: for example, Bloomberg reported that a 48% correct patch rate was associated with "very positive" feedback and a general "helpful" opinion [93, p. 5].

We augment this analysis with an independent assessment from Yang *et al.*, an established expert team in APR [24], [25], [26], [27], [28], who analyzed the semantics of the produced repairs against the human-written patches and found 14 of the produced repairs to be semantically identical to the human patches (see Table 3). While APR expertise is not equivalent to domain expertise, APR experts tend to be more suited to assessing the patches produced by these methods due to "creative" (or adversarial or potentially-overfitting) nature of such patches [94], [95], [96], and evidence suggests that domain-experts may not be a strong gold standard [97]. We acknowledge that this assessment is not a substitute for a full human study on patch correctness; however, having two independent teams find converging results adds confidence that a majority of the plausible repairs do not overfit to the testbench (a common problem in APR for software [50], [98], [99]), since we inspect intermediate wire values when assigning fitness scores. We do note that correctness is critical in hardware designs (e.g., since manufactured chips cannot be easily updated once deployed), and our use case does not involve deploying patches directly but instead showing plausible patches to developers to reduce maintenance costs [35], [36].

4. 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 [92], but we discuss the synthesizability of CirFix in practice in Section 7.

```
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
```

Fig. 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.

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 [11], and only recently have there been works investigating multi-edit repairs [90], [100]. 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 with multiple edits to the faulty circuit design. We return to multi-edit repairs in the human study (Section 6).

Fault Localization Sensitivity. To assess repair performance as fault localization quality decreases, we conducted a targeted experiment reducing the quality of the initial fault location available to CirFix in a controlled manner. This sort of investigation, in which the sensitivity of the algorithm with respect to fault localization is assessed, is important is software APR [101], [102], [103], [104].

When simulation outputs are compared against expected behavior to produce the initial set of wires and registers with mismatched values (see Section 3.1), we also randomly include some correct wires and registers (with probability 25%, 50%, or 75%) as "noise". Because our fault localization is a transitive fixed point calculation, additional initial elements may result in larger fault localization sets (e.g., informally, the traditional scalability problem with fault localization is that almost everything may end up implicated).

We focus on defect scenarios CirFix successfully re-

TABLE 4
Repair results for CirFix with added noise to initial mismatch set for our fault localization algorithm. "Defect Cat." indicates the category for the defect, "Normalized Repair Time" shows the normalized time for repair (in seconds) when compared to the original repair, and a '—' indicates no repair was found in 5 independent trials. "Noise" indicates the percent of disturbance placed on the fault localization. The ordering of the benchmarks follows Table 3.

Project	Defect Cat.	Normalized Repair Time 25% Noise	50% Noise	75% Noise
decoder_3_to_8	1	1.11×	—	—
counter	1	0.49×	0.45×	0.05×
	1	0.48×	0.58×	0.86×
	1	0.06×	0.98×	0.98×
flip_flop	1	0.99×	0.38×	1.86×
	1	0.87×	1.18×	0.35×
fsm_full	1	0.77×	0.08×	0.58×
	2	0.58×	0.57×	0.81×
	2	3.21×	3.24×	1.76×
lshift_reg	1	1.07×	0.11×	0.11×
	1	0.18×	0.49×	0.21×
	1	1.01×	0.32×	0.60×
mux_4_1	1	0.27×	0.35×	0.61×
	2	1.19×	1.27×	1.24×
i2c	2	0.93×	0.39×	0.34×
	2	0.04×	0.13×	0.13×
	2	18.57×	—	15.88×
sha3	1	1.44×	2.80×	3.60×
	2	0.67×	0.33×	0.73×
reed_solomon_ decoder	2	1.39×	0.52×	1.29×
sdram_ controller	2	0.11×	1.22×	0.55×

paired. Table 4 presents normalized results of five trials at each noise level. Of the 21 defect scenarios CirFix originally plausibly repaired, CirFix also found plausible repairs for all 21 when subjected to 25% noise, 19 at 50% noise, and 20 at 75% noise. Execution times with lower-quality fault localization are not statistically different to those found without fault localization noise ($p = 0.7$, $p = 0.6$, $p = 0.9$, unpaired Student t-test), suggesting that CirFix performs similarly even if the design or testbench does not admit precise fault localization. Any difference in execution times can be attributed to the randomness of the search for repairs (a larger fault localization set may result in new candidate repairs or repairs being considered in a different order). An investigation of this outcome reveals that many of the same registers and wires were transitively implicated in both cases (i.e., with and without noise). For example, in the largest benchmark (reed_solomon_decoder), there are 10 (out of 11 maximum) elements in the initial mismatch set and 114 in the final fault localization set. With 75% noise, there are 11 elements in the initial set but 124 in the final fault localization set. This small increase suggests that many of the potential wires and registers were already transitively implicated without the added noise. Our targeted experiment furthers confidence that CirFix's novel fault localization approach scales to larger designs or those with more complicated or less precise testbenches that do not admit accurate initial fault localization.

> 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. Lastly, our sensitivity investigation gives confidence that CirFix's fault localization approach scales to larger designs.

5.2 RQ2. 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 [44]). 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.

6 Evaluation of Human Study

Next, we present statistical analyses of the responses to our human study. In total, 41 users participated in our survey

and each completed 6 debugging tasks. We consider the following additional research questions:

RQ3. Does CirFix's fault localization algorithm improve designers' objective performances?

RQ4. In what contexts do designers find CirFix's fault localization algorithm helpful?

6.1 RQ3. Fault Localization and Human Performance

We assessed programmer performance by evaluating (1) F-scores (F_1) of correctly-identified faults for each debugging task by each participant and (2) total time taken to complete a debugging task within no specific time limit (see Section 4.3). A participant is said to correctly identify faults for a given defect scenario if they identified program line(s) that contain a bug or missing line(s). F-scores were evaluated by calculating the harmonic mean of recall and precision.

To evaluate the statistical significance of participants utilizing the fault localization as a debugging aid as opposed to none, we used the unpaired Student t-test. We did not observe a statistically-significant difference in time taken to localize faults with full or no annotations from our fault localization ($p = 0.41$). On average, participants spent 299.6 seconds with full annotations as opposed to 239.0 seconds with no annotations. A participant with an F-score of 1 correctly identified faulty program line(s) or missing line(s) in the defect scenario, while a F-score of 0 meant no faulty program line(s) or missing line(s) were correctly identified. We did find that the objective F-score for participants given full localization was higher ($F_1 = 0.67$) than the objective F-score for participants who had half fault localization ($F_1 = 0.33$), which in turn was higher than those without fault localization ($F_1 = 0.29$). However, this trend did not rise to the level of statistical significance ($p = 0.12$). We predict that the results indicate CirFix data-flow based notion of fault localization can be a useful tool for manual debugging.

In addition, we found statistically-significant differences in the F-scores between experts ($F_1 = 0.37$) and novices ($F_1 = 0.17$) when they had CirFix's fault localization with a large effect size ($p = 0.04$, $d = 0.54$). This statistic did not survive correcting for multiple comparisons. However, all other significant values reported survive correcting for multiple comparisons ($q = 0.05$) to avoid false discovery. We used Cohen's d due to similarities in standard deviations in the groups.

> CirFix fault localization produced no significant improvement in designer's objective performance.

6.2 RQ4. Subjective Judgment of Fault Localization

We assessed participant subjective judgements of CirFix's fault localization support in various contexts, including debugging multi-line defects and different circuit designs (see Section 4.1).

For each presented stimulus with a debugging aid, participants were asked to rate, on a Likert scale, the usefulness and accuracy of the tool in helping them localize the circuit defects as seen on Figure 4. Differences in the number of responses per rating arise because not all participants answered all questions.

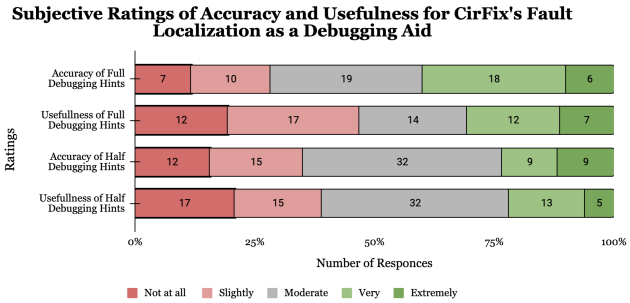

Fig. 4. A visual representation of the distribution of ratings subjects gave to CirFix's fault localization when viewed as a debugging aid. Subjects rated the tool as a debugging aid based on accuracy and usefulness on a scale of 1–5, where 1 represents not at all accurate or useful and 5 represents extremely accurate or useful.

Participants rated full fault localization support on student-developed designs to be significantly more useful and accurate than full support for open source projects ($p = 0.01$, $d = 0.7$; $p = 0.002$, $d = 1.05$, a large effect size). These results suggest our algorithm would be more beneficial for debugging in pedagogical environments.

Most interestingly, we find that participants rated CirFix's fault localization support to be significantly more useful and accurate for debugging multi-line defects than single-line defects with a large effect size ($p = 0.002$, $d = 1.04$; $p = 0.003$, $d = 0.86$). Given that support for multi-line software repairs is limited [105], [106], with most tools only supporting single-line repairs, our results, by contrast, are promising for reducing maintenance costs associated with more complex defects in the hardware domain.

> CirFix fault localization is may be significantly helpful for multi-line defects ($p = 0.002$) in pedagogical contexts.

6.3 Human Study Discussion

CirFix includes two novel approaches that we hypothesize contribute to its success in repairing hardware defects: the fault localization algorithm and the fitness function. When coupled with prior direct study of this fitness function [23, Sec. 5.3] that found the function to be highly effective at guiding the search for repairs, our human study helps tease apart the two components. In particular, we conclude that the fitness function is critical to CirFix's overall success (e.g., its high fitness distance correlation was shown in Section 5.2). By contrast, the fault localization algorithm is more useful for humans in particular contexts, such as multi-line defects.

There have been previous concerns about ranked list fault localization [107], [108] and the degree to which humans make effective use of multiple implicated lines. Although not directly comparable, our fault localization both does not utilize ranked lists and also does well in multi-line contexts. We believe our success suggests promising future directions.

Furthermore, the statistically significant results on the subjective judgment of CirFix's fault localization may prove to be more beneficial for pedagogy. In our qualitative analysis of optional questions given to participants at the end of

the study, we found that participants, particularly novices, who self-reported to be less effective at tasks related to debugging hardware designs, rated the debugging features (e.g., highlighting of implicated statements and naming of implicated wires or registers) to be significantly useful ($p = 0.02$, $d = 0.97$; $p = 0.001$, $d = 1.35$). This indicates that debugging aids with supplemental supportive features, such as our fault localization algorithm, could help novices navigate these tasks. Despite advances in hardware development platforms, novices still report intimidation by circuitry [109]. The self-efficacy of students can be improved by providing them with support they find useful, such as our fault localization algorithm.

7 Discussion of Synthesizability and Timing

Professional hardware designers often aim to construct a physical system that passes all tests in the real world. We consider two ways a design may fail to complete that end-to-end process: synthesizability and timing constraints.

Synthesizable designs are defined as descriptions that can generate a physical system (e.g., ASIC) using a pre-defined set of basic building blocks (see IEEE 1364.1 [58]). Synthesizability centers on avoiding certain language constructs (e.g., force and release, or fork and join, which are mainly used for simulation purposes) that cannot translate into physical circuits and may also depend on the electronic design automation tools used. Because the line between synthesizable and non-synthesizable designs is nuanced [110], [111], designers may be instructed to follow established guidelines [58], [112]. For example, it can be difficult to synthesize module instances that initiate a delay on built-in gates [113].

Timing constraints center on whether or not the circuit converges to produce the correct answer in time. Informally, signals must propagate and converge along the "critical path" to an output within a given budget or frequency [114]. For example, a circuit design that is synthesized with a 10nm process and meets all timing constraints may not behave correctly if it is instead fabricated with a larger, slower 45nm process. Similarly, a design that meets timing constraints when first fabricated but is then changed by inserting additional delays on its critical path may then fail to meet those previous constraints, e.g., FPGA timing errors that may arise during the "place and route" step after synthesis [115].

Since CirFix produces fixes to faulty hardware designs ignoring plausible synthesizability or timing constraint changes, we consider all 21 patches from Table 3 and manually examined 18 to assess changes in the design that may negatively impact the end-to-end process. We exclude 3 patches that repaired non-synthesizable designs not appropriate for timing constraints. We found that no patches introduced specific constructs that are characterized to lead to non-synthesizability. In addition, we found 9/18 of the patches to contain changes (such as adding delays along a critical path) that we infer may impact the predefined timing budgets in the design.

Because our notion of synthesizability is based solely on structural elements of the Verilog design that can be detected statically, a modified version of CirFix that avoids introducing those elements would increase confidence that if a design was synthesizable before CirFix, then it would remain synthesizable after being patched by CirFix. Similarly, CirFix might make use of static timing analysis (STA) or worst case execution time (WCET) calculations and reject edits that may slow the design. Unfortunately, however, such static analyses of circuit designs are often inaccurate or conservative (e.g., [116]). As a result, we expect that practitioners would still carry out simulations, waveform analyses, and post-fabrication testing to authenticate the viability of a CirFix-patched system.

8 Limitations and Threats to Validity

Our results 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 [117].

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 [118].

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).

While our results on the scalability of CirFix's repairs gives us confidence that our implementation scales to larger benchmarks than those we tested, additional developer effort may be needed to apply CirFix to very large designs, such as modularizing the design and testbench (cf. functions and unit tests in software). We leave further optimizations to the CirFix fault localization (e.g., more efficient pruning of the search space) as future work.

Finally, our human study participants are students. While they may represent new hires joining the workforce, they are not indicative of experienced hardware designers.

9 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.* [6] and Ran *et al.* [7] employ software analysis approaches to identify statements in design code responsible for defects, but suffer from high false positive rates.

Bloem and Wotawa [8] use formal analysis of circuit descriptions to identify defects, and Peischl and Wotawa [14] use a model-based diagnosis paradigm that supports source-level debugging of large VHDL designs at the statement and expression level. This use of formal methods for error diagnoses is orthogonal to our work, but could be applied to reduce the search space for approaches like CirFix.

Staber *et al.* [67] 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 larger, real-world hardware descriptions. Chang *et al.* [9] explicitly insert multiplexers to automatically diagnose faults in hardware designs and suggest repairs; Madre *et al.* [10] 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 [11], [12], [13]. 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 [22], [32], [43], [49]. Semantics-driven approaches first extract constraints on a program based on test suite execution and then use these constraints to synthesize a patch [63], [90], [119], [120]. 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 parallel hardware descriptions.

10 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, since spectrum-based approaches commonly used in APR do not apply to hardware designs. 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. Lastly, we evaluated the relative utility of our novel fault localization algorithm independent of our automated repair context via a human study. We found a statistically significant preference ($p = 0.003$) for CirFix fault localization as a debugging aid in fixing multi-line defects, primarily in student applications ($p = 0.01$).

References

[1] F. Schirrmeister, M. McNamara, L. Melling, and N. Bhatnagar, "Debugging at the hardware/software interface," June 2012. [Online]. Available: https://www.embedded-computing.com/embedded-computing-design/debugging-at-the-hardware-software-interface

[2] H. Foster, "Assertion-based verification: Industry myths to realities (invited tutorial)," in *International Conference on Computer Aided Verification*. Springer, 2008, pp. 5–10.

[3] J. Wagner, "Intel could make billions off of meltdown & spectre," Feb 2018. [Online]. Available: https://www.digitaltrends.com/computing/intel-could-make-billions-off-meltdown-spectre/

[4] D. Athow, "Pentium fdiv: The processor bug that shook the world," Oct 2014. [Online]. Available: https://www.techradar.com/news/computing-components/processors/pentium-fdiv-the-processor-bug-that-shook-the-world-1270773

[5] R. Lemos, "Intel releases fix for f00f bug," Nov 1997. [Online]. Available: https://www.zdnet.com/article/intel-releases-fix-for-f00f-bug/

[6] T.-Y. Jiang, C.-N. Liu, and J. Y. Jou, "Estimating likelihood of correctness for error candidates to assist debugging faulty hdl designs," in *2005 IEEE International Symposium on Circuits and Systems*. IEEE, 2005, pp. 5682–5685.

[7] J.-C. Ran, Y.-Y. Chang, and C.-H. Lin, "An efficient mechanism for debugging RTL description," in *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, 2003. Proceedings.* IEEE, 2003, pp. 370–373.

[8] R. Bloem and F. Wotawa, "Verification and fault localization for vhdl programs," *Journal of the Telematics Engineering Society (TIV)*, vol. 2, pp. 30–33, 2002.

[9] K.-h. Chang, I. Wagner, V. Bertacco, and I. L. Markov, "Automatic error diagnosis and correction for rtl designs," in *2007 IEEE International High Level Design Validation and Test Workshop*. IEEE, 2007, pp. 65–72.

[10] J. C. Madre, O. Coudert, and J. P. Billon, "Automating the diagnosis and the rectification of design errors with priam," in *1989 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, 1989, pp. 30–33.

[11] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2017.

[12] M. Monperrus, "The living review on automated program repair," HAL/archives-ouvertes.fr, Tech. Rep. hal-01956501, 2018.

[13] Y. Liu, L. Zhang, and Z. Zhang, "A survey of test based automatic program repair." *JSW*, vol. 13, no. 8, pp. 437–452, 2018.

[14] B. Peischl and F. Wotawa, "Automated source-level error localization in hardware designs," *IEEE Design & Test of Computers*, vol. 23, no. 1, pp. 8–19, 2006.

[15] G. Friedrich, M. Stumptner, and F. Wotawa, "Model-based diagnosis of hardware designs," *Artificial Intelligence*, vol. 111, no. 1, pp. 3–39, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S000437029900034X

[16] F. Wotawa, "Using multiple models for debugging vhdl designs," in *Proceedings of the 14th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: Engineering of Intelligent Systems*, ser. IEA/AIE '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 125–134.

[17] ——, "On the relationship between model-based debugging and program slicing," *Artificial Intelligence*, vol. 135, no. 1, pp. 125–143, 2002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370201001618

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

[19] Y. Huang, H. Ahmad, S. Forrest, and W. Weimer, "Applying automated program repair to dataflow programming languages," in *GI @ ICSE 2021*, J. Petke, B. R. Bruce, Y. Huang, A. Blot, W. Weimer, and W. B. Langdon, Eds. internet: IEEE, 30 May 2021.

[20] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. IEEE, 1995, pp. 143–151.

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

[22] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "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, 2012, pp. 3–13.

[23] H. Ahmad, Y. Huang, and W. Weimer, "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*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 990–1003. [Online]. Available: https://doi.org/10.1145/3503222.3507763

[24] D. Yang, Y. Qi, and X. Mao, "Evaluating the strategies of statement selection in automated program repair," in *International Conference on Software Analysis, Testing, and Evolution*. Springer, 2018, pp. 33–48.

[25] A. Guo, X. Mao, D. Yang, and S. Wang, "An empirical study on the effect of dynamic slicing on automated program repair efficiency," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 554–558.

[26] D. Yang, K. Liu, D. Kim, A. Koyuncu, K. Kim, H. Tian, Y. Lei, X. Mao, J. Klein, and T. F. Bissyandé, "Where were the repair ingredients for defects4j bugs?" *Empirical Software Engineering*, vol. 26, no. 6, pp. 1–33, 2021.

[27] D. Yang, Y. Qi, X. Mao, and Y. Lei, "Evaluating the usage of fault localization in automated program repair: an empirical study," *Frontiers of Computer Science*, vol. 15, no. 1, pp. 1–15, 2021.

[28] D. Yang, Y. Lei, X. Mao, D. Lo, H. Xie, and M. Yan, "Is the ground truth really accurate? dataset purification for automated program repair," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 96–107.

[29] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 559–572.

[30] "Chapter 6 - the case for synchronous design," in *Top-Down Digital VLSI Design*, H. Kaeslin, Ed. Boston: Morgan Kaufmann, 2015, pp. 357–389. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B978012800730300006X

[31] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.

[32] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, 2011, pp. 1427–1434.

[33] A. Zeller, "Automated debugging: Are we close," *Computer*, no. 11, pp. 26–31, 2001.

[34] L. Bening and H. Foster, "Rtl formal verification," *Principles of Verifiable RTL Design: A functional coding style supporting verification processes in Verilog*, pp. 103–129, 2001.

[35] W. Weimer, "Patches as better bug reports," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, ser. GPCE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 181–190. [Online]. Available: https://doi.org/10.1145/1173706.1173734

[36] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, "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*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 75–87. [Online]. Available: https://doi.org/10.1145/3395363.3397351

[37] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software quality journal*, vol. 21, no. 3, pp. 421–443, 2013.

[38] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 2002, pp. 467–477.

[39] S. Sudakrishnan, J. Madhavan, E. J. Whitehead Jr, and J. Renau, "Understanding bug fix patterns in verilog," in *Proceedings of the 2008 international working conference on Mining software repositories*, 2008, pp. 39–42.

[40] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 273–282.

[41] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[42] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[43] Z. Qi, F. Long, S. Achour, and M. Rinard, "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*, 2015, pp. 24–36.

[44] T. Jones and S. Forrest, "Fitness distance correlation as a measure of problem difficulty for genetic algorithms," in *ICGA*, vol. 95, 1995, pp. 184–192.

[45] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 802–811.

[46] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[47] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "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, 2019, pp. 456–467.

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

[49] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 254–265.

[50] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[51] R. Poli and W. B. Langdon, "Genetic programming with one-point crossover," in *Soft Computing in Engineering Design and Manufacturing*. Springer, 1998, pp. 180–189.

[52] B. L. Miller and D. E. Goldberg, "Genetic algorithms, selection schemes, and the varying effects of noise," *Evolutionary computation*, vol. 4, no. 2, pp. 113–131, 1996.

[53] C. Le Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, 2012, pp. 959–966.

[54] C. S. Timperley, "Advanced techniques for search-based program repair," Ph.D. dissertation, University of York, 2017.

[55] J. Vasconcelos, J. A. Ramirez, R. Takahashi, and R. Saldanha, "Improvements in genetic algorithms," *IEEE Transactions on magnetics*, vol. 37, no. 5, pp. 3414–3417, 2001.

[56] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, 2009, pp. 947–954.

[57] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.

[58] IEEE, "Ieee standard for verilog hardware description language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–590, 2006.

[59] S. Romano, C. Vendome, G. Scanniello, and D. Poshyvanyk, "A multi-study investigation into dead code," *IEEE Transactions on Software Engineering*, 2018.

[60] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2015, pp. 451–460.

[61] V. Synopsys, "Verilog simulator," *Avaliable HTTP: http://www. synopsys. com/products/simulation/simulation. html*, 2004.

[62] Synopsys, "VCS functional verification solution," 2020. [Online]. Available: https://www.synopsys.com/verification/simulation/vcs.html

[63] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[64] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest, "Software mutational robustness," *Genetic Programming and Evolvable Machines*, vol. 15, no. 3, pp. 281–312, 2014.

[65] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[66] R. Just, D. Jalali, and M. D. Ernst, "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*, 2014, pp. 437–440.

[67] S. Staber, B. Jobstmann, and R. Bloem, "Finding and fixing faults," in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 2005, pp. 35–49.

[68] P. McMinn, "Search-based failure discovery using testability transformations to generate pseudo-oracles," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, 2009, pp. 1689–1696.

[69] R. Feldt, "Generating diverse software versions with genetic programming: an experimental study," *IEE Proceedings-Software*, vol. 145, no. 6, pp. 228–236, 1998.

[70] L. I. Manolache and D. G. Kourie, "Software testing using model programs," *Software: Practice and Experience*, vol. 31, no. 13, pp. 1211–1236, 2001.

[71] S. R. Shahamiri, W. M. N. W. Kadir, S. Ibrahim, and S. Z. M. Hashim, "An automated framework for software test oracle," *Information and Software Technology*, vol. 53, no. 7, pp. 774–788, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584911000589

[72] K. Aggarwal, Y. Singh, A. Kaur, and O. Sangwan, "A neural net based approach to test oracle," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 3, pp. 1–6, 2004.

[73] F. Gholami, N. Attar, H. Haghighi, M. V. Asl, M. Valueian, and S. Mohamadyari, "A classifier-based test oracle for embedded software," in *2018 Real-Time and Embedded Systems and Technologies (RTEST)*, 2018, pp. 104–111.

[74] S. Hertz, D. Sheridan, and S. Vasudevan, "Mining hardware assertions with guidance from static analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 6, pp. 952–965, 2013.

[75] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, B. Xie, and H. Mei, "Supporting oracle construction via static analysis," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 178–189.

[76] M. Hanafy, H. Said, and A. M. Wahba, "New methodology for digital design properties extraction from simulation traces," in *2015 Tenth International Conference on Computer Engineering Systems (ICCES)*, 2015, pp. 91–98.

[77] S. R. Shahamiri, W. M. N. W. Kadir, and S. Z. Mohd-Hashim, "A comparative study on automated software test oracle methods," in *2009 Fourth International Conference on Software Engineering Advances*, 2009, pp. 140–145.

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

[79] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "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*, 2010, pp. 182–191.

[80] S. Afshan, P. McMinn, and M. Stevenson, "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*, 2013, pp. 352–361.

[81] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "Test oracle assessment and improvement," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 247–258.

[82] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[83] C. L. Seitz, C. Mead, and L. Conway, "System timing," *Introduction to VLSI systems*, pp. 218–262, 1980.

[84] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[85] N. Alsolami, Q. Obeidat, and M. Alenezi, "Empirical analysis of object-oriented software test suite evolution," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 11, 2019.

[86] B. L. Miller, D. E. Goldberg *et al.*, "Genetic algorithms, tournament selection, and the effects of noise," *Complex systems*, vol. 9, no. 3, pp. 193–212, 1995.

[87] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, 2010.

[88] Z. Sharafi, I. Bertram, M. Flanagan, and W. Weimer, "Eyes on code: A study on developers code navigation strategies," *IEEE Transactions on Software Engineering*, 2020.

[89] S. Stapleton, Y. Gambhir, A. LeClair, Z. Eberhart, W. Weimer, K. Leach, and Y. Huang, "A human study of comprehension and code summarization," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 2–13.

[90] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[91] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. T. Sullivan, W. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard, "Automatically patching errors in deployed software," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, J. N. Matthews and T. E. Anderson, Eds. ACM, 2009, pp. 87–102. [Online]. Available: https://doi.org/10.1145/1629575.1629585

[92] V. Taraate, *Digital logic design using verilog: coding and RTL synthesis*. Springer, 2016.

[93] S. Kirbas, E. Windels, O. McBello, K. Kells, M. Pagano, R. Szalanski, V. Nowack, E. R. Winter, S. Counsell, D. Bowes, T. Hall, S. Haraldsson, and J. Woodward, "On the introduction of automatic program repair in Bloomberg," *IEEE Software*, vol. 38, no. 4, pp. 43–51, 2021.

[94] Y. Huang, K. Leach, Z. Sharafi, N. McKay, T. Santander, and W. Weimer, "Biases and differences in code review using medical imaging and eye-tracking: Genders, humans, and machines," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 456–468. [Online]. Available: https://doi.org/10.1145/3368089.3409681

[95] J. Lehman, J. Clune, D. Misevic, C. Adami, L. Altenberg, J. Beaulieu, P. J. Bentley, S. Bernard, G. Beslon, D. M. Bryson, N. Cheney, P. Chrabaszcz, A. Cully, S. Doncieux, F. C. Dyer, K. O. Ellefsen, R. Feldt, S. Fischer, S. Forrest, A. Fundefinedenoy, C. Gagńe, L. Le Goff, L. M. Grabowski,

B. Hodjat, F. Hutter, L. Keller, C. Knibbe, P. Krcah, R. E. Lenski, H. Lipson, R. MacCurdy, C. Maestre, R. Miikkulainen, S. Mitri, D. E. Moriarty, J.-B. Mouret, A. Nguyen, C. Ofria, M. Parizeau, D. Parsons, R. T. Pennock, W. F. Punch, T. S. Ray, M. Schoenauer, E. Schulte, K. Sims, K. O. Stanley, F. Taddei, D. Tarapore, S. Thibault, R. Watson, W. Weimer, and J. Yosinski, "The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities," *Artif. Life*, vol. 26, no. 2, p. 274–306, may 2020. [Online]. Available: https://doi.org/10.1162/artl_a_00319

[96] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus, "How to design a program repair bot? insights from the repairnator project," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2018, pp. 95–104.

[97] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 26–36. [Online]. Available: https://doi.org/10.1145/2025113.2025121

[98] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "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*, 2015, pp. 532–543.

[99] X. B. D. Le, F. Thung, D. Lo, and C. Le Goues, "Overfitting in semantics-based automated program repair," *Empirical Software Engineering*, vol. 23, no. 5, pp. 3007–3033, 2018.

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

[101] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, "A critical review on the evaluation of automated program repair systems," *Journal of Systems and Software*, vol. 171, p. 110817, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302156

[102] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 102–113.

[103] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 416–426.

[104] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[105] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing evolution for multi-hunk program repair," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 13–24. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00020

[106] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 691–701.

[107] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 199–209. [Online]. Available: https://doi.org/10.1145/2001420.2001445

[108] X. Xia, L. Bao, D. Lo, and S. Li, ""automated debugging considered harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 267–278.

[109] F. Anderson, T. Grossman, and G. Fitzmaurice, "Trigger-action-circuits: Leveraging generative design to enable novices to design and build circuitry," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 331–342. [Online]. Available: https://doi.org/10.1145/3126594.3126637

[110] J. Gillenwater, G. Malecha, C. Salama, J. Grundy, and J. O'Leary, "Formalizing and enhancing verilog," 01 2007.

[111] C. Salama, J. Gillenwater, G. Malecha, A. Zhu, W. Taha, J. Grundy, and J. O'Leary, "Synthesizable verilog," 01 2007.

[112] J. Gillenwater, G. Malecha, C. Salama, A. Y. Zhu, W. Taha, J. Grundy, and J. O'Leary, "Synthesizable high level hardware descriptions: Using statically typed two-level languages to guarantee verilog synthesizability," in *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ser. PEPM '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 41–50. [Online]. Available: https://doi.org/10.1145/1328408.1328416

[113] H. Bhatnagar, *Advanced ASIC chip synthesis using Synopsys® Design compiler™ Physical compiler™ and Primetime®*. Springer US, 2002.

[114] K. Morris, "Timing is everything: The trouble with timing closure in FPGA design," *Electronic Engineering Journal*, May 2013.

[115] R. Aggarwal, "Fpga place & route challenges," in *Proceedings of the 2014 on International symposium on physical design*, 2014, pp. 45–46.

[116] S. Simoglou, C. Sotiriou, and N. Blias, "Static timing analysis induced simulation errors for asynchronous circuits," in *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2021, pp. 1–4.

[117] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "Hardfails: Insights into software-exploitable hardware bugs," in *USENIX Security Symposium*, 2019, pp. 213–230.

[118] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *International Symposium on Search Based Software Engineering*. Springer, 2011, pp. 33–47.

[119] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 448–458.

[120] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of ISSTA*, 2016.