RTL-Repair: Fast Symbolic Repair of Hardware Design Code

Kevin Laeufer
laeufer@eecs.berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

Brandon Fajardo*
brfajardo@berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

Abhik Ahuja*
ahujaabhik@berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

Vighnesh Iyer
vighnesh.iyer@eecs.berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

Borivoje Nikolić
bora@eecs.berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

Koushik Sen
ksen@eecs.berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

ABSTRACT

We present RTL-Repair, a semantics-based repair tool for register transfer level circuit descriptions. Compared to the previous state-of-the-art tool, RTL-Repair generates more correct repairs within seconds instead of minutes or even hours. We imagine that RTL-Repair could thus be integrated into an IDE to give developers repair suggestions promptly. Our new SMT-based one-step fault localization and repair algorithm for digital hardware designs uses optimization to generate minimal changes that a user can easily understand. A novel adaptive windowing approach allows us to avoid scalability issues by focusing the repair search on the parts of the test that matter the most. RTL-Repair provides repairs that pass their testbench for 9 out of 12 real bugs collected from open-source hardware projects. Two repairs fully match the ground truth, one partially, four more repairs change the correct expression but in a way that overfits the testbench, and only three repairs differ strongly from the ground truth.

CCS CONCEPTS

• **Hardware** → *Semi-formal verification*; *Hardware description languages and compilation*; • **Software and its engineering** → *Search-based software engineering*.

ACM Reference Format:
Kevin Laeufer, Brandon Fajardo, Abhik Ahuja, Vighnesh Iyer, Borivoje Nikolić, and Koushik Sen. 2024. RTL-Repair: Fast Symbolic Repair of Hardware Design Code. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620666.3651346

1 INTRODUCTION

Modern digital circuits are commonly designed at the register transfer level (RTL). At this level of abstraction, a designer uses a hardware description language (HDL) to specify how registers and memories in the circuit are updated each cycle. Synthesis tools convert the RTL description into a low-level circuit, which can be implemented in a VLSI physical design flow.

Most logic bugs in a hardware design manifest at the RTL level. For testing, RTL descriptions can be executed by a wide range of available simulators [20, 25, 41, 43, 47]. While some designers only use manual inspection of the resulting waveforms to validate the output [30], more rigorous testing setups use verification libraries allowing self-checking tests [4, 18, 23, 28, 45]. Besides concrete testing, formal techniques like bounded-model checking [11] can reveal inputs to the design that cause assertion violations. No matter whether checks are manual, automated, or symbolic, for each bug, the designer is always presented with a failing trace of signal values over time that leads to a violation.

The classic way of debugging this failing trace is for the designer to look at a rendering of it and use their knowledge of the design to try and find a way to fix it. Some recent academic work has also looked into source-level debugging for hardware languages [50], but traditional debugging tools from software, such as step-through debugging, are not always as useful in the hardware context. While software programmers are used to thinking of their programs as executing strictly in program order, multi-threaded programs break this abstraction, which makes them much more challenging to reason about. In hardware, there are no sequential programs. Results and state updates are all computed concurrently. HDLs reflect that by modeling components as a composition of parallel processes (in Verilog [2] and VHDL [3]) or by allowing signals to be used before they are assigned (last-connect semantics in Chisel [8]). RTL-Repair sidesteps this problem by directly suggesting relevant changes to the RTL developer.

The software engineering community has long been working on automated program repair [22, 27, 29, 34]. In the standard scenario, we are provided with program source code and test cases, at least one of which currently fails. The tool then tries to find one or several changes to the source code, which makes all test cases pass. Unfortunately, most automated program repair tools take several hours to run and often provide unsatisfying repairs, which remove

*Denotes equal contribution.

Table 1: RTL-Repair vs State-of-the-Art Tool

	RTL-Repair			CirFix [6]		
	#	median	max	#	median	max
✔ Correct Repairs	16	0.70s	13.17s	10	2.53min	14.19h
✖ Wrong Repairs	2	0.51s	0.68s	11	2.03h	9.50h
○ Cannot Repair	14	5.64s	59.81s	11	16.00h	16.00h

program functionality [39]. Recent work on a tool called CirFix shows that automated program repair can be applied to hardware descriptions as well [6]. However, CirFix can take several hours to come up with a repair and often results in unsatisfactory repairs.

In this paper, we present RTL-Repair, which produces more correct repairs than CirFix in a fraction of the time (Table 1). We demonstrate how to combine the repair template idea from CirFix with symbolic analysis-based repair and how to address scalability issues associated with long-running testbenches. RTL-Repair is available on github: https://github.com/ekiwi/rtl-repair. We also provide an artifact with scripts to reproduce all our results. Our paper makes the following contributions:

- We propose a new symbolic, template-based repair algorithm
- We introduce an adaptive windowing technique that allows us to scale to long-running testbenches
- We define a new output/state divergence delta (OSDD) metric that helps reason about the hardness of bugs
- We perform a thorough evaluation of RTL-Repair and CirFix, including gate-level simulation as a new way to automatically verify repairs of hardware
- We further evaluate RTL-Repair on real bugs mined from open-source projects [31]

2 BACKGROUND

2.1 The SystemVerilog Language

Most digital hardware today is designed with hardware description languages (HDLs) like VHDL [3] or SystemVerilog [2]. Both languages were initially conceived to program event-driven simulations of hardware designs [19] and then later adapted to support synthesis. Synthesis tools that automatically translate a chip simulation into a netlist to be fabricated are standard practice in the industry nowadays.

Synthesizability. Not all simulation constructs have a mapping to actual hardware, which leads to the definition of a synthesizable subset of the language [1, 42]. The mix of simulation language and automated translation can complicate hardware design: Circuits that seem to work well in simulation might fail to synthesize. A much more severe problem is synthesis-simulation mismatch, where a design is quietly accepted by the synthesis tool, but the resulting hardware behaves differently from the high-level HDL description [35]. Standard approaches to detect simulation-synthesis mismatch are combinational equivalence checking [26], which attempts to prove equivalence between the high-level RTL and the low-level netlist and gate-level simulations [19].

X-Propagation. In a Verilog program, most values are 4-state bit-vectors: each bit can take on a value of 0, 1, Z, or X. The Z value is used to model tri-state buses. The X value is used to model unknown values. For example, these can originate from uninitialized state variables, out-of-bounds reads, unconnected signals, or explicit assignments of a signal to X [46]. Simulation with X values could be thought of as abstract interpretation since an X can stand for both 0 and 1. However, in Verilog, execution with X values is neither sound nor complete, meaning that for some computations with X, the result is over-approximated, and for others, it is under-approximated. Especially the over-approximation, also known as X-optimism, can lead to a mismatch between the 4-state simulation and the 2-state circuit generated by the synthesis tool. X-propagation is thus a common source of synthesis-simulation mismatch.

2.2 Bounded Model Checking.

While simulator-based dynamic verification executes a Verilog design with concrete input values, formal verification reasons about all possible input values at once to prove or disprove a specification. Bounded model checking (BMC) is a popular formal technique to find bugs in hardware designs [11]: Starting from an arbitrary state, it unrolls the system for k cycles and asks a formal engine based on SAT [33] or SMT [10] if there exist any inputs, and starting state which will make an assertion in the design fail. The result is either the assurance that all assertions hold for up to k cycles or a trace that shows how the assertion can fail. Bounded model checking generally gets slower as k increases, often exponentially so.

3 REPAIR EXAMPLE

To illustrate key components of the RTL-Repair algorithm, we present an example before diving into details in Section 4. We are going to repair the Verilog description of a simple counter circuit (Figure 1a). This is the same example circuit that was used in the CirFix paper [6]. We first illustrate how the circuit can be converted to perform BMC with an SMT-solver before showing how RTL-Repair adapts BMC for its repair algorithm.

Transition System Encoding. Before we can formally analyze the circuit, we need to convert the Verilog code into a format that is

```
module first_counter (
   input clock, input reset, input enable,
   output reg [3:0] count,
   output reg overflow
);
always@(posedge clock) begin
 if(reset == 1'b1) begin
   // count reset is missing:
   // count <= 4'b0;
   overflow <= 1'b0;
 end else if(enable == 1'b1) begin
   count <= count + 1;
 end
 if(count == 4'b1111) begin
   overflow <= 1'b1;
 end
end
endmodule
```

(a) Verilog source code.

```
overflow' =
 (ite (= count (  bv15 4)) true
 (ite reset false overflow))
count' =
 (ite reset count
 (ite enable (bvadd count (_ bv1 32))
```

(b) Next state expressions in SMTLib format.

Figure 1: A counter circuit with a missing reset value.

reset	*enable*	*count*	*overflow*
1	X	X	X
0	0	0	X

(a) I/O Trace Generated from a Testbench.

```
module first counter (
   // [...] I/O from original circuit
   input φ0, input [3:0] α0,
   input φ1, input [3:0] α1);
always @(posedge clock) begin
 if(reset == 1'b1) begin
   overflow <= 1'b0;
   if(φ0) count <= α0;
 end else if(enable == 1'b1) begin
   count <= count + 1;
 end
 if(count == 4'b1111) begin
   overflow <= 1'b1;
   if(φ1) count <= α1;
 end
end
endmodule
```

(b) Simplified conditional overwrite template applied.

```
; random concrete initial state
(assert (= overflow@0 true))
(assert (= count@0 (_ bv8 4)))
; next state
(define-fun count@1 () ( BitVec 4)
 (ite (and (= count@0 ( bv15 4)) φ1) α1
 (ite (and reset@0 φ0) α0
 (ite (and (not reset@0) enable@0)
   (bvadd count@0 (_ bv1 4)) count@0))))
; I/O trace
(assert reset@0)
(assert (= count@1 ( bv0 4)))
; limit number of changes to one
(assert (= #b01 (bvadd
  (ite phi1 #b01 #b00)
  (ite phi0 #b01 #b00))))
```

(c) Repair query.

ϕ_0	α_0	ϕ_1	α_1	change size: $\sum_i^{N=2} \phi_i$
1	0	0	X	1
1	0	1	0	2

(d) Possible solutions.

Figure 2: Repairing the counter circuit from Figure 1

amenable to formal analysis. We use the open-source synthesis tool yosys [48] to turn the event-driven simulation into a circuit-like transition system representation that encodes the clock updates for the `count` and `overflow` registers as SMTLib [9] bit-vector expressions (Figure 1b).

I/O Trace. The bug we are looking to fix is revealed by a simple test: After we reset the circuit, we expect the `count` output to be zero. However, currently, it is X since the `count` register is missing a reset assignment. RTL-Repair accepts concrete tests in the form of I/O traces – essentially tables with one row for every execution cycle and one column for every input and expected output value. Figure 2a shows the trace for our small example test. Besides manual entry, an I/O trace can be recorded from a concrete testbench, similar to how CirFix obtains expected outputs for its fitness function. It could also be returned by a BMC tool that has discovered a bug in the circuit. We designate inputs that could be set to any value with X. For outputs, an X indicates that the value of the output at that particular time step does not matter, i.e., it is not checked by the testbench.

Repair Template. RTL-Repair analyses the Verilog source code and enumerates all possible changes to the circuit that fit a certain template. In our example, we consider assigning a constant to a signal somewhere in code. For each assignment, we create two new inputs: ϕ_i and α_i. α_i represents a constant that can be freely chosen by the repair synthesizer. ϕ_i indicates whether the assignment should be included. Figure 2b shows how we add two possible new assignments to the circuit. Generally, we will add a lot more possible assignments, however, we restrict ourselves to two in this example in order to make the resulting synthesis query easy to understand. For a more thorough description of the various repair templates, please see Section 4.2. The instrumented Verilog AST is converted into a transition system using yosys [48].

The Basic Repair Synthesizer. RTL-Repair unrolls the transition system exactly the same way as we would for bounded model checking. However, instead of asking the solver to choose the inputs, we assert that the input and output values are equal to the ones from the given I/O trace and ask the solver to provide an assignment to our synthesis variables ϕ_i and α_i such that the circuit correctly follows the I/O trace. Such a repair query is shown in Figure 2c. Figure 2d shows two solutions found by the solver. Both solutions add an assignment to the reset block. The difference is that the second solution also adds an assignment in the overflow code block. The I/O trace never increments the counter all the way to 15, which makes this new statement dead code in terms of the test we provided. However, assigning `count` to 0 in the overflow block, as the solver suggests, introduces a new bug in our circuit which is revealed if we test the overflow behavior. In general, we find that the fewer changes we make, the more likely we will arrive at a valid repair. We thus implement an algorithm that ensures a repair with a minimal number of changes.

First, we use the solver to check whether a solution with any number of changes exists at all. If that is the case, we then search for a solution with a minimal number of changes by calculating $\sum_i^N \phi_i$ in the SMT query and successively increasing the number of changes we want to see until the solver returns a satisfying assignment. This constraint is demonstrated at the bottom of Figure 2c. By restricting the number of changes to one, we obtain the minimal solution with $\phi_0 = 1$ and $\phi_1 = 0$. While simple, the major downside of the basic repair synthesis approach is that we always unroll the system for all cycles in the I/O trace. This leads to scalability issues with long testbenches, which can be solved by our new adaptive windowing approach described in Section 4.4.

Repairing the Verilog Code. We use the repair synthesizer's assignment to the synthesis variables to generate the repaired Verilog code. We remove any assignment where ϕ_i is false. This can be thought of as plugging in the assignment from the synthesizer and running a simple dead-code elimination. We inline the concrete value for α_i for all remaining code, where ϕ_i is true, and thus, the assignment happens unconditionally. After making these changes on the AST, we serialize it into a repaired Verilog file.

4 THE RTL-REPAIR REPAIR ALGORITHM

The RTL-Repair tool accepts a buggy Verilog module as well as an I/O trace as input. It first runs a standard static analysis tool to address some straightforward errors that would lead to non-synthesizable code. Next RTL-Repair applies a series of repair templates, each of which is implemented as a compiler pass over the Verilog AST which adds different ways for the repair synthesizer to fix the circuits. Our repair synthesizer takes the transition system (converted from Verilog with yosys) and the testbench in the form of an I/O trace as input. It then tries to find a minimal change from the space of changes described by the repair template that will make the circuit pass the test. If such a minimal change is found, it is applied to the Verilog AST, resulting in a repaired source code. If the change is large ($\sum_i^N \phi_i > 3$), then we keep on trying out templates to see if a smaller repair can be found with a different template. If no change can make the I/O trace pass, RTL-Repair will move on to the next repair template. Once all repair templates have been tried with no success, the user is notified that no repair could be found. The whole process is illustrated in Figure 3.

4.1 Preprocessing with Static Analysis

RTL-Repair's symbolic repair algorithm requires the buggy design to be synthesizable [1, 42]. This is generally not a problem. In industry, static analysis tools called linters are used to enforce coding standards that guarantee that the circuit can be synthesized. Modern hardware languages like Chisel allow users only to express synthesizable circuits [8]. A study of bugs in open-source hardware projects did not find any issues with synthesizability in practice [31]. However, novices might still make these kinds of mistakes in Verilog, and they are pervasive in the benchmarks targeted by CirFix [6]. Thus, we employ the open-source Verilog simulator Verilator as a linter [41] to deal with two common issues that prevent a circuit from being synthesizable.

Blocking and Non-Blocking Assignments. Synthesizable Verilog for synchronous circuits generally consists of two different kinds of processes: Ones that describe combinational logic, marked by a sense list that triggers re-computation on the change of any signal, and processes triggered by clock events that describe synchronous logic (registers and memories). By convention, combinational processes use blocking assignments, and synchronous logic processes use non-blocking assignments [17]. If the linter warns about the wrong kind of assignment, we automatically change it to the appropriate version depending on the type of process to ensure correct synthesis with yosys [48].

Latches. are state elements that get updated when their input changes. In modern ASIC technologies, latches are generally disallowed in favor of clock edge-triggered flip-flops. Latches also cannot be represented in the transition system format used by RTL-Repair. Many Verilog beginners will unintentionally write code that describes a latch instead of the combinational logic they intended to encode because of a missing assignment in a process. We remove any latches in the Verilog description by providing a default value for a signal whenever there is a warning from the static analysis tool about a latch. We use zero as a default value since it is always valid to assign regardless of the bit-width of the signal. If needed for a repair, the default can be overwritten by the Replace Literals repair template introduced in the next section.

4.2 Repair Templates

A repair template is a compiler pass that analyzes the Verilog AST and adds a range of possible changes, thus describing a space of possible repairs for the repair synthesizer. Each change is guarded by an indicator variable ϕ_i, which will disable the change when set to zero. Besides that, many templates introduce additional free variables α_i, which represent constants in the Verilog code that the repair synthesizer can freely choose. If $\sum_i^N \phi_i = 0$, we turn off all changes and obtain the original circuit. The repair synthesizer will try to find an assignment to all synthesis variables ϕ_i and α_i that makes the circuit obey the I/O trace subject to $min \sum_i^N \phi_i$. Minimizing the number of changes has two advantages: (1) it ensures that the synthesizer does not change code that is not relevant to the given I/O trace, making it more likely that the fix will generalize to other tests (2) the smaller the suggested repair, the easier it will be for a developer to verify. We have developed three different repair templates which are able to fix a wide range of bugs. In our framework, new repair templates can be easily added without any changes to the repair synthesizer as long as they use ϕ_i and α_i variables as described above.

Our symbolic repair templates are inspired by the templates used by CirFix [6]. However, while applying a CirFix template will produce a single concrete change to the RTL description, an application of a RTL-Repair template encodes a large set of changes that the synthesizer can choose from. Concretely, while CirFix's *Conditional* repair template will pick a single random conditional to invert, our *Add Guard* template will present the synthesizer with the possibility to invert every single condition in the RTL

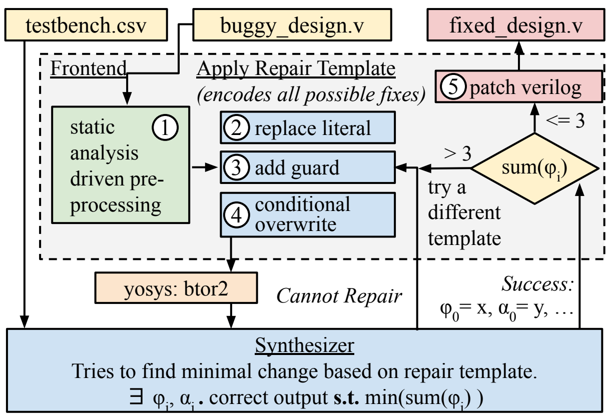


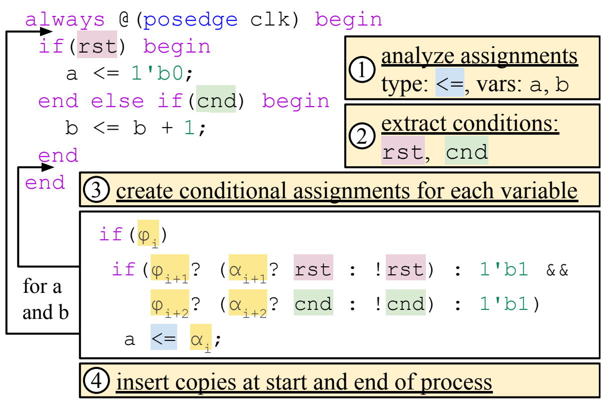
Figure 3: RTL-Repair Flow

description. Our templates are thus much more powerful compared to CirFix's, which explains why three templates are enough for RTL-Repair to solve a large number of benchmarks.

Replace Literals Template. This template allows the repair synthesizer to replace literal integer values with a freely chosen constant. In order to ensure that we obtain a synthesizable circuit, we restrict the integer literals that can be replaced with the ones appearing in r-value expressions. Thus we exclude integer literals that specify signal types (bit-width), parameters, and any other integer literals that cannot be replaced with a non-constant expression. Figure 6 shows some examples of integer literals that can and cannot be replaced.

Add Guard Template. This template allows the repair synthesizer to invert or add a guard to the condition of any if-statement or the right-hand side of any 1-bit assignment in the circuit. The transform follows this template $e \rightarrow (\neg?)e \wedge ((\neg?)a(\vee(\neg?)b)?)$, where e is the original expression, $\neg?$ indicates an optional negation and $(\vee(\neg?)b)?$ an optional second part of the guard. The cost of inverting e is one, the cost of adding a simple guard $\wedge a$ is one, and the cost of adding a more complex guard $\wedge(a \vee b)$ is two. For a and b, the synthesizer is able to pick from a list of 1-bit variables that are part of the circuit. Care has to be taken not to create new combinational loops in the circuit since that would prevent us from synthesizing it. We thus first calculate all combinational dependencies in the original input circuit and then restrict a and b to variables that won't create any new dependencies for the left-hand side of the assignment. Figure 5 demonstrates our conservative approach with an example.

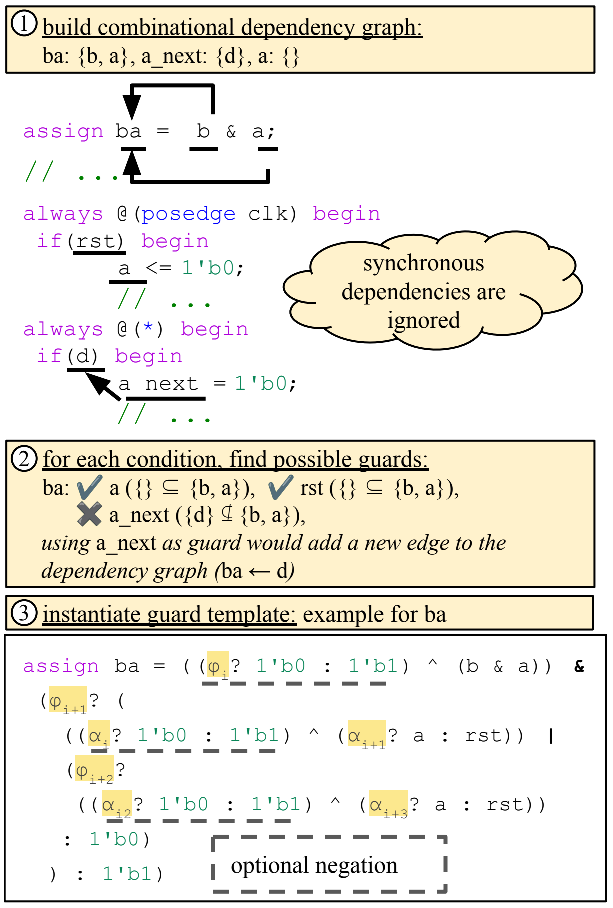
Conditional Overwrite Template. This template allows the repair synthesizer to insert new assignments of a freely chosen constant value to any signal. These assignments can happen either at the start or the end of a process and can optionally be guarded. The guard is composed of conditions extracted from the same process. Figure 4 shows an example. The cost of adding an unconditional assignment is one (α_i in Figure 4). Each guard within the assignment has an


```
always @(posedge clk) begin
  if(rst) begin
    a <= 1'b0;
  end else if(cnd) begin
    b <= b + 1;
  end
end
```

① analyze assignments
type: <=, vars: a, b
② extract conditions:
rst, cnd
③ create conditional assignments for each variable
for a and b
④ insert copies at start and end of process

```
if(φi)
  if(φi+1? (αi+1? rst : !rst) : 1'b1 &&
     φi+2? (αi+2? cnd : !cnd) : 1'b1)
    a <= αi;
```

Figure 4: Conditional Overwrite Template: Allows the repair synthesizer to assign every variable to an arbitrary constant at the start and end of every process. This assignment can be guarded by conditions mined from the same process.



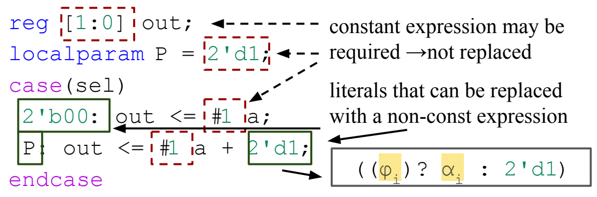
① build combinational dependency graph:
ba: {b, a}, a_next: {d}, a: {}

```
assign ba =  b & a;
// ...
always @(posedge clk) begin
 if(rst) begin
    a <= 1'b0;
    // ...
always @(*) begin
 if(d) begin
    a_next = 1'b0;
    // ...
```

synchronous dependencies are ignored
② for each condition, find possible guards:
ba: ✓ a ({} ⊆ {b, a}), ✓ rst ({} ⊆ {b, a}),
✗ a_next ({d} ⊈ {b, a}),
using a_next as guard would add a new edge to the dependency graph (ba ← d)
③ instantiate guard template: example for ba

```
assign ba = ((φi? 1'b0 : 1'b1) ^ (b & a)) &
 (φi+1? (
   ((αi? 1'b0 : 1'b1) ^ (αi+1? a : rst)) |
   (φi+2?
    ((αi2? 1'b0 : 1'b1) ^ (αi+3? a : rst))
   : 1'b0)
  ) : 1'b1)
```

Figure 5: Add Guard Template: Allows the synthesizer to append a guard to certain 1-bit expressions. We conservatively choose possible guards to ensure that no combinational cycles are created and synthesizability is maintained.


```
reg [1:0] out;
localparam P = 2'd1;
case(sel)
  2'b00: out <= #1 a;
  P: out <= #1 a + 2'd1;
endcase
```

constant expression may be required →not replaced
literals that can be replaced with a non-const expression
((φi)? αi : 2'd1)

Figure 6: Replace Literals Template: Conservatively replaces literals in places where the expression is not required to evaluate to a constant at compile time.

additional cost of one (α_{i+1} and α_{i+2}). To maintain synthesizability, our compiler pass first analyzes each process to determine which signals are assigned to in it and whether it uses blocking or non-blocking assignments. This is necessary since assigning the same signal from multiple different processes leads to race conditions in the Verilog simulation and is thus undesirable. We also want to maintain the invariant that only one type of assignment is used throughout a single process.

4.3 Basic Synthesizer

We first discuss a very basic version of our synthesizer. The next section covers how adaptive windowing can help us scale to larger benchmarks.

Inputs. Our synthesizer takes in a circuit design with synthesis variables ϕ_i and α_i from the application of a repair template as well as a testbench. The design is provided in the btor2 format, which is obtained by running the synthesis tool yosys on the Verilog code. This step will fail if the design is not synthesizable. The testbench is in the format of a table with rows for each cycle of execution and columns for each input and output signal of the circuit that we are trying to repair.

Unknown Values. All registers start out uninitialized, and some input signals might not be defined in certain cycles of the test execution because the testbench author did not consider their value to be relevant for the test. These unknown values are modeled with Xs in Verilog. In our synthesis procedure, we either randomize or set unknown values to zero. We chose to randomize when the original testbench was using X values, as is the case for all benchmarks from CirFix. We set unknown to zero if the original testbench was using Verilator to match the behavior of that simulator.

Synthesizing a Repair. Having chosen concrete values for initial states and unspecified inputs, we unroll the circuit and assert that inputs and outputs have the values assigned to them from the testbench while keeping the synthesis variables ϕ_i and α_i symbolic. Then we query the SMT-solver to obtain an assignment to the synthesis variables that will make the testbench pass. If the solver returns unsatisfiable, we know that the given template cannot repair the circuit, and move on to the next template. If the solver returns a solution, we try to minimize the number of changes.

Synthesizing a Minimal Repair. We observed in our experiment that when a solution exists, the minimal solution generally only takes a small number of changes (see Table 5). We thus start searching for a minimal solution in a linear search, starting with one change. We encode the number of changes as a constraint into our SMT query. If a solution exists, we found a minimal solution which is returned to the frontend to repair the Verilog code. If the solver returns unsatisfiable, we increase the number of expected changes by one. This optimization can be framed as an instance of the Max-SMT problem [12], however, most solvers that perform well on hardware circuits do not implement Max-SMT directly. We thus stick with our customized algorithm allowing us to use a wide range of specialized SMT solvers.

4.4 Adaptive Windowing

As part of our basic synthesis process, we need to unroll the system once for every cycle in the testbench execution. Unfortunately, bounded model checking, and thus also our synthesis algorithm, scales purely with the number of times we unroll the system. Adaptive windowing allows us to synthesize repairs while unrolling the system for only a small number of cycles. We observed that human developers often start investigating a bug by looking at the signal values around the cycle where the first violation occurred. To make debugging tractable, developers may assume that the state of the circuit a couple of cycles before the bug manifests is correct, as this makes it simpler to trace the signal values to find a reason for the divergence. We can make use of this assumption to reduce the scope of our unrolling.

We define two values: k_{past} and k_{future}, which specify how many cycles before and after the first output divergence we unroll our system. Our algorithm starts with both values set to zero. Thus, in the first iteration, our tool concretely executes the original circuit until the step at which the output divergence occurred and then starts the symbolic unrolling from the concrete state reached after those steps. If all state update functions are correct and there is only a bug in how the output is computed from current state and inputs, then this would be enough to obtain a correct repair.

We generally sample all minimal repairs for a given k_{past} and k_{future} and then evaluate them through a concrete simulation using the repaired circuit. If the test passes, we have found a correct repair which we return from the synthesizer. If none of the repairs work, we analyze their failures. If all of them failed at or before the same cycle as the original failure, then we assume that some state update in the past went wrong, and we need to increase the symbolic execution window towards the past. We thus increment k_{past} by a constant. Generally, we chose step size two. If, on the other hand, there exists a repair that makes the earlier failure go away but then leads to a failure later in the circuit execution, we assume that we are missing some future context, and we, therefore, increase k_{future} such that our repair window includes the newly failing cycle. The window size is the sum of k_{past} and k_{future}. In our RTL-Repair implementation, we set the maximum window size to 32, after which the tool will give up and declare that it cannot find a repair. We also observed that when there are many failing repairs, it generally pays off to go to a larger window size immediately. Our implementation thus advances to the next window sizer after finding four failing repairs.

We have found this new adaptive windowing technique to improve scalability for benchmarks with longer testbenches drastically. One benchmark, in particular, went from timing out after one hour to being repaired in less than ten seconds.

5 OUTPUT / STATE DIVERGENCE DELTA

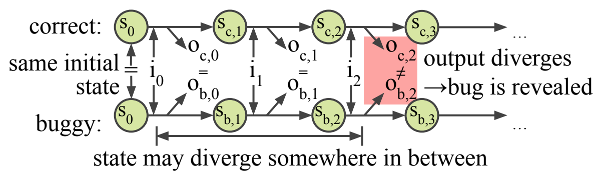
We formalize the insight behind our adaptive windowing technique through the output/state divergence delta (OSDD) metric. We assume that we are provided with a working digital synchronous circuit (the ground truth), a buggy version of the same design, a sequence of test inputs, and a starting assignment to all state variables. We then calculate the OSDD by comparing outputs and state variables on every cycle of the test execution. We note the distance between the first divergence in state values and the first divergence in output values. If the state never diverges, then the OSDD is zero. Otherwise, the OSDD is the number of steps from when the state first diverges to when the output diverges, plus one. An illustrated example of this is shown in Figure 7. This definition requires that the state and output variables are the same between the buggy and ground-truth versions which is true for all benchmarks that can be correctly repaired by RTL-Repair and CirFix.

Table 2: Output / State Divergence Delta Evaluation: Testbench (TB) length in cycles, first error (output divergence), output/state divergence delta (OSDD), size of the repair window used by RTL-Repair as well as repair results. Two i2c benchmarks are excluded as they are not clocked, and thus, the OSDD is not defined.

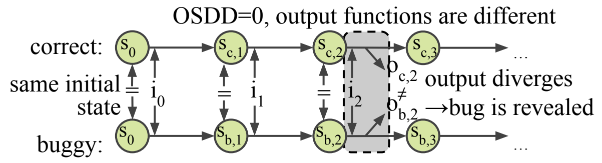
Benchmark	TB Cycles	First Error	OSDD	Window	RTL-Repair	CirFix
decoder_w1	28	0	0	[0 .. 10]	✔	✖
decoder_w2	28	0	0	[0 .. 20]	✖	○
counter_w1	27	4	n/a		○	✔
counter_k1	26	3	1	[-2 .. 0]	✔	✔
counter_w2	26	19	1	[-2 .. 0]	✔	✔
flop_w1	11	0	1	[-1 .. 0]	✔	✔
flop_w2	11	0	1	[-2 .. 1]	✔	✔
fsm_w1	37	32	1		○	○
fsm_s2	37	9	1		✔	✖
fsm_w2	37	2	3		✔	✖
fsm_s1	37	10	11		✔	✖
shift_w1	27	8	1		✔	✖
shift_w2	27	0	1	[-1 .. 0]	✔	✔
shift_k1	29	7	n/a		✖	✔
mux_k1	151	10	1		○	○
mux_w2	151	20	1	[0 .. 10]	✔	✖
mux_w1	151	10	1	[0 .. 20]	✔	✖
i2c_k1	171957	1238	13	[-4 .. 0]	✔	✔
sha3_w1	357	24	1		○	✔
sha3_r1	357	24	1		○	○
sha3_w2	357	46	n/a		○	○
sha3_s1	129	31	1	[-2 .. 0]	✔	✖
pairing_w1	74149	74119	73346		○	○
pairing_k1	74149	775	2		○	○
pairing_w2	74149	74119	74109		○	○
reed_b1	166166	2967	2963		○	○
reed_o1	166166	0	0		○	✖
sdram_w2	636	130	1	[-4 .. 5]	✔	○
sdram_k2	636	64	25		✔	○
sdram_w1	636	1	1		○	✖

We empirically calculated the OSDD by discretizing the testbench waveforms and extracting output and state (register) information from the synthesized netlist of the circuits. Our results are shown in Table 2. The benchmark with the largest OSDD that was successfully repaired is sdram_k2, with an OSDD of 25. However, our static analysis-based preprocessing step solved this benchmark, which does not require any unrollings (see Table 5). The next benchmark is i2c_k1 with an OSDD of 13, which was actually solved by the unrolling-based repair synthesizer. Both RTL-Repair and CirFix were only able to solve benchmarks with low OSDD. High OSDD benchmarks are difficult because both tools try to reason about the execution of the system.

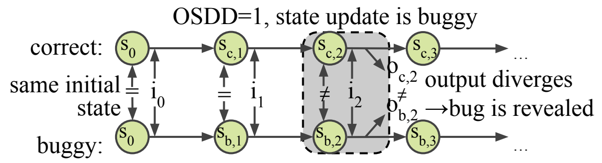
For all benchmarks repaired by RTL-Repair's synthesis engine, the OSDD provides a lower bound for how far the repair window needs to be expanded into the past. The i2c_k1 benchmark only requires a repair window of size 4, which is lower than the OSDD.



(a) We start the correct and the buggy system in the same state and execute both with the same inputs until the outputs become unequal, which happens in our example after two state updates.



(b) If all states before the output divergence are equivalent, then we define the OSDD to be zero.



(c) OSDD=1

Figure 7: Output / State Divergence Delta (OSDD) Example

While the buggy register value diverges already 12 cycles before the bug manifests, it is also updated only four cycles in the past, allowing our repair synthesizer to generate the correct repair with only 4 cycles of context. Other benchmarks require larger repair windows because future information needs to be taken into account. For example, the decoder benchmarks contain no state variables, and their OSDD is 0. However, several different inputs and, therefore, several cycles of test execution are needed in order to reveal all the bugs in the design.

6 EVALUATION

We compare RTL-Repair to the prior state-of-the-art tool CirFix in terms of the quality of repairs and how quickly the repairs are provided. RTL-Repair provides more correct repairs and is often orders of magnitude faster than CirFix. We also performed a detailed analysis of the various components of RTL-Repair and how they contribute to its performance.

6.1 Experimental Setup

All our experiments were run on a server with 252GiB of RAM and two 8-core Intel Xeon E5-2667 CPUs with hyperthreading. While the core algorithms of both RTL-Repair and CirFix could benefit from multiple cores, their current implementations are strictly sequential, and thus, multiple cores are only used to run different benchmarks in parallel to speed up our evaluation. We observed that CirFix would run slower on our machine than reported in the original paper. This could be due to the slower CPU, or VCS might have higher startup costs on our machine due to a different license server setup. We increased the timeout from 12h to 16h to ensure that CirFix has the time to generate all repairs reported by the original paper.

The RTL-Repair prototype consists of a frontend that uses the PyVerilog [44] library to implement our symbolic repair templates. The Yosys [48] tool converts Verilog designs into a transition system in the btor2 format [38]. A synthesis engine written in Rust takes the I/O trace and transition system to find a suitable repair. While the synthesis engine can work with many different SMT solvers, we use bitwuzla [37] in our experiments since it offers the best performance on average.

We extended the CirFix prototype [6] to allow us to run different benchmarks in parallel in order to speed up the evaluation. The core algorithm remains untouched, and our results are comparable to those reported in the CirFix paper. Table 3 shows the benchmarks from the CirFix paper that are used in our evaluation and maps

Table 3: Benchmark Overview. Relates benchmarks from CirFix [6] to the short names used throughout this paper.

Project	Defect	Short Name
decoder 3-8	Two separate numeric errors	decoder_w1
	Incorrect assignment	decoder_w2
counter	Incorrect sensitivity list	counter_w1
	Incorrect reset	counter_k1
	Incorrect incremental of counter	counter_w2
flip flop	Incorrect conditional	flop_w1
	Branches of if-statement swapped	flop_w2
fsm full	Incorrect case statement	fsm_w1
	Incorrectly blocking assignments	fsm_s2
	Assignment to next state and default in case statement omitted	fsm_w2
	Assignment to next state omitted, incorrect sensitivity list	fsm_s1
lshift reg	Incorrect blocking assignment	shift_w1
	Incorrect conditional	shift_w2
	Incorrect sensitivity list	shift_k1
mux 4 1	1 bit instead of 4 bit output	mux_k1
	Hex instead of binary constants	mux_w2
	Three separate numeric errors	mux_w1
i2c	Incorrect sensitivity list	i2c_w1
	Incorrect address assignment	i2c_w2
	No command acknowledgement	i2c_k1
sha3	Off-by-one error in loop	sha3_w1
	Incorrect bitwise negation	sha3_r1
	Incorrect assignment to wires	sha3_w2
	Skipped buffer overflow check	sha3_s1
tate pairing	Incorrect logic for bitshifting	pairing_w1
	Incorrect operator for bitshifting	pairing_k1
	Incorrect instantiation of modules	pairing_w2
reed-solomon decoder	Insufficient register size	reed_b1
	Incorrect sensitivity list for reset	reed_o1
sdram-controller	Numeric error in definitions	sdram_w2
	Incorrect case statement	sdram_k2
	Incorrect assignments to registers during synchronous reset	sdram_w1

Table 4: Repair Correctness Evaluation
Symbols: ✔ test passed, ✖ test failed, ○ no repair to test
An empty cell means that the test did not apply. Overall a repair is judged a success (✔) if all applicable tests pass. The number in the right-most column denotes the number of changes comprising the repair.

Benchmark	Tool	Tool Status	Testbench	CirFix Author	Gate-Level	iVerilog	Extended	Overall
decoder_w1	rtlrepair	✔	✔		✔		✔	2 ✔
	cirfix	✔	✔	✔	✔		✖	3 ✖
decoder_w2	rtlrepair	✔	✔		✔		✖	5 ✖
	cirfix	○						○
counter_w1	rtlrepair	○						○
	cirfix	✔	✔	✔	✔	✔		1 ✔
counter_k1	rtlrepair	✔	✔		✔	✔		1 ✔
	cirfix	✔	✔	✔	✔	✔		5 ✔
counter_w2	rtlrepair	✔	✔		✔	✔		2 ✔
	cirfix	✔	✔	✔	✔	✔		1 ✔
flop_w1	rtlrepair	✔	✔		✔	✔		0 ✔
	cirfix	✔	✔	✔	✔	✔		1 ✔
flop_w2	rtlrepair	✔	✔		✔	✔		0 ✔
	cirfix	✔	✔	✔	✔	✔		2 ✔
fsm_s2	rtlrepair	✔	✔		✔	✔		15 ✔
	cirfix	✔	✔	✖	✖	✔		2 ✖
fsm_w2	rtlrepair	✔	✔		✔	✔		3 ✔
	cirfix	✔	✔	✖	✖	✔		1 ✖
fsm_s1	rtlrepair	✔	✔		✔	✔		2 ✔
	cirfix	✔	✔	✔	✖	✔		1 ✖
shift_w1	rtlrepair	✔	✔		✔	✔		4 ✔
	cirfix	✔	✔	✔	✔	✖		1 ✖
shift_w2	rtlrepair	✔	✔		✔	✔		0 ✔
	cirfix	✔	✔	✔	✔	✔		1 ✔
shift_k1	rtlrepair	✔	✖		✖	✖		0 ✖
	cirfix	✔	✔	✔	✔	✔		1 ✔
mux_w2	rtlrepair	✔	✔		✔	✔		2 ✔
	cirfix	✔	✔	✖	✖	✔		3 ✖
mux_w1	rtlrepair	✔	✔		✔	✔		9 ✔
	cirfix	✔	✔	✖	✖	✔		4 ✖
i2c_w1	rtlrepair	○						○
	cirfix	✔	✔	✔				1 ✔
i2c_w2	rtlrepair	○						○
	cirfix	✔	✔	✖				1 ✖
i2c_k1	rtlrepair	✔	✔					1 ✔
	cirfix	✔	✔	✔				1 ✔
sha3_w1	rtlrepair	○						○
	cirfix	✔	✔	✔	✔	✔		1 ✔
sha3_s1	rtlrepair	✔	✔		✔	✔		1 ✔
	cirfix	✔	✔	✔	✖	✔		1 ✖
reed_o1	rtlrepair	○						○
	cirfix	✔	✔	✔	✖	✔		2 ✖
sdram_w2	rtlrepair	✔	✔			✔		2 ✔
	cirfix	○						○
sdram_k2	rtlrepair	✔	✔			✔		2 ✔
	cirfix	○						○
sdram_w1	rtlrepair	○						○
	cirfix	✔	✔	✔		✖		2 ✖

decoder_w1: Two separate numeric errors

diff original vs. bug

```
- ({en,A,B,C} == 4'b1010)? 8'b1111 1011 :
+ ({en,A,B,C} == 4'b1000)? 8'b1111 1011 :
  ({en,A,B,C} == 4'b1011)? 8'b1111 0111 :
  ({en,A,B,C} == 4'b1100)? 8'b1110 1111 :
  ({en,A,B,C} == 4'b1101)? 8'b1101 1111 :
  ({en,A,B,C} == 4'b1110)? 8'b1011 1111 :
  ({en,A,B,C} == 4'b1111)? 8'b0111 1111 :
-                          8'b1111 1111;
+                          8'b0111_1111;
```

diff bug vs. our repair

```
- ({en,A,B,C} == 4'b1000)? 8'b1111 1011 :
+ ({en,A,B,C} == 4'b1010)? 8'b1111 1011 :
  ({en,A,B,C} == 4'b1011)? 8'b1111 0111 :
  ({en,A,B,C} == 4'b1100)? 8'b1110 1111 :
  ({en,A,B,C} == 4'b1101)? 8'b1101 1111 :
  ({en,A,B,C} == 4'b1110)? 8'b1011 1111 :
  ({en,A,B,C} == 4'b1111)? 8'b0111 1111 :
-                          8'b0111 1111;
+                          8'b1111_1111;
```

RTL-Repair (0.4s, Replace Literals): Max-SMT guarantees a minimal number of changes in the solution and thus no untested functionality is changed.

diff bug vs. CirFix repair

```
- ({en,A,B,C} == 4'b1000)? 8'b1111 1011 :
+ ({en,A,A,C} == 4'b1000)? 8'b1111 1011 :
- ({en,A,B,C} == 4'b1011)? 8'b1111 0111 :
+ ({en,A,B,C-1}==4'b1011)? 8'b1111 0111 :
  ({en,A,B,C} == 4'b1100)? 8'b1110 1111 :
  ({en,A,B,C} == 4'b1101)? 8'b1101 1111 :
  ({en,A,B,C} == 4'b1110)? 8'b1011 1111 :
- ({en,A,B,C} == 4'b1111)? 8'b0111 1111 :
-                          8'b0111_1111;
+ (C - 1);
```

CirFix (7h): repair passes testbench, but changes code that is never tested.

counter_w1: Incorrect sensitivity list

diff original vs. bug

```
- always @(posedge clk) begin : COUNTER
+ always @(clk) begin : COUNTER
```

RTL-Repair (0.9s): Cannot find a repair. Removing the `posedge` fundamentally changes the synthesized circuit, turning a process describing registers (state elements) into a process describing a purely combinatorial circuit. Since the repair synthesizer works directly on the synthesized circuit, it cannot reason about this bug.

diff bug vs. CirFix repair

```
- always @(clk) begin : COUNTER
+ always @(posedge clk) begin : COUNTER
```

CirFix (35s): has a matching template that adds a `posedge` to a random process. This benchmark features only a single process.

sdram_w1: Incorrect assignments to registers during synchronous reset

diff original vs. bug

```
  always @ (posedge clk)
   if (~rst_n) begin
     [...]
-     wr data r <= 1'b0;
-     rd data r <= 1'b0;
+     rd_data_r <= data_in;
```

diff bug vs. CirFix repair

```
  always @ (posedge clk)
   if (~rst_n) begin
     [...]
+     rd data r <= IDLE;
+     state_cnt_next = 4'd0;
```

CirFix (7h): correctly adds back the reset for `rd_data_r` (`IDLE` is 0). In the ground truth circuit, the reset value of `wr_data_r` is never read and thus unnecessary. The assignment to `state_cnt_next` creates a race condition.

diff bug vs. our repair

```
  always @ (posedge clk)
   [...]
+  if(!rst_n) rd_data_r <= 16'b0;
```

RTL-Repair (1.5min, Basic Synth, Conditional Overwrite): the conditional overwrite template correctly generates a minimal repair. However, the adaptive windowing algorithm gives up too soon and the repair is only found by the more precise but much slower basic synthesizer if we increase the timeout from 60s to 90s.

sha3_s1: Skipped buffer overflow check

diff original vs. bug

```
- assign update = (accept | (state & (~buffer full))) & (~done);
+ assign update = (accept | (state)) & (~done);
```

diff bug vs. CirFix repair

```
- always @(posedge clk)
+ always @(*)
   if (reset) done <= 0;
   else if (state & out_ready) done <= 1;
```

CirFix (1.6min): changes `done` from a register to a latch. While this works to fix the bug in simulation, it creates unwanted synthesis-simulation mismatch.

diff bug vs. our repair

```
- assign update = (accept | (state)) & (~done);
+ assign update = (accept | (state)) & (~done) & (~f_ack);
```

RTL-Repair (4s, Add Guard): proposes a simple change to the correct expression while maintaining a circuit that synthesizes correctly. A better testbench would be needed in order to distinguish between this repair and the ground truth.

Figure 8: Qualitative comparison of RTL-Repair and CirFix repairs on four different benchmarks. The decoder_w1 result shows how the Max-SMT-based approach can help RTL-Repair generate repairs that leave untested features untouched, while CirFix sometimes introduces new bugs. counter_w1 is a good example for a bug that RTL-Repair cannot tackle because it leads to synthesis problems. sdram_w1 shows how RTL-Repair avoids introducing new bugs by minimizing repairs.

Table 5: Repair Speed Evaluation. A direct comparison of RTL-Repair and CirFix is on the right, and the performance breakdown of the RTL-Repair components is on the left. A bold number indicates the number of changes performed. The Basic Synthesizer column shows the performance of RTL-Repair when benchmarks are naively unrolled without our adaptive windowing technique. Symbols: ✔ generated correct repair, ✖ generated incorrect repair, ○ no repair generated

Benchmark	Preprocessing	Replace Literals	Add Guard	Conditional Overwrite	Basic Synthesizer	RTL-Repair	CirFix	Speedup
decoder_w1	0 0.16s	2 ✔ 0.18s	○ 0.09s	○ 0.09s	✔ 0.41s	✔ 0.39s	✖ 7.21h	66,904x
decoder_w2	0 0.18s	5 ✖ 0.26s	○ 0.10s	○ 0.09s	✖ 0.59s	✖ 0.68s	Timeout	85,149x
counter_w1	6 0.44s	○ 0.11s	○ 0.13s	○ 0.13s	○ 0.83s	○ 0.83s	✔ 35.09s	42x
counter_k1	0 0.18s	○ 0.12s	○ 0.08s	1 ✔ 0.09s	✔ 0.65s	✔ 0.60s	✔ 13.31h	**79,945x**
counter_w2	0 0.17s	○ 0.20s	○ 0.19s	2 ✔ 0.10s	✔ 0.67s	✔ 0.75s	✔ 14.19h	**67,978x**
flop_w1	0 0.18s	○ 0.11s	1 ✔ 0.11s	○ 0.10s	✔ 0.45s	✔ 0.45s	✔ 15.28s	**34x**
flop_w2	0 0.17s	○ 0.11s	2 ✔ 0.11s	○ 0.10s	✔ 0.44s	✔ 0.43s	✔ 28.57min	**3,961x**
fsm_w1	0 0.17s	○ 0.96s	○ 1.44s	○ 2.65s	○ 1.26s	○ 5.76s	Timeout	10,007x
fsm_s2	15 0.46s	Repaired by preprocessing			✔ 0.66s	✔ 0.65s	✖ 2.03h	11,191x
fsm_w2	3 0.70s	Repaired by preprocessing			✔ 0.90s	✔ 0.90s	✖ 44.83min	2,990x
fsm_s1	2 0.67s	Repaired by preprocessing			✔ 0.90s	✔ 0.90s	✖ 1.11min	73x
shift_w1	4 0.43s	Repaired by preprocessing			✔ 0.58s	✔ 0.57s	✖ 28.58s	50x
shift_w2	0 0.16s	○ 0.13s	1 ✔ 0.10s	○ 0.11s	✔ 0.52s	✔ 0.46s	✔ 35.11s	**75x**
shift_k1	0 0.17s	○ 0.12s			✖ 0.34s	✖ 0.34s	✔ 15.51s	45x
mux_k1	4 0.79s	○ 0.12s	○ 0.08s	○ 0.14s	✖ 1.15s	○ 1.28s	Timeout	44,840x
mux_w2	0 0.17s	2 ✔ 0.13s	○ 0.08s	○ 0.09s	✔ 0.47s	✔ 0.36s	✖ 5.42h	54,731x
mux_w1	6 0.58s	3 ✔ 0.10s	○ 0.06s	○ 0.15s	✔ 0.86s	✔ 0.81s	✖ 7.56h	33,542x
i2c_w1	1 0.85s	○ 0.18s	○ 0.46s	○ 0.31s	○ 1.82s	○ 1.90s	✔ 3.86min	122x
i2c_w2	1 0.84s	○ 0.19s	○ 0.41s	○ 0.29s	○ 1.70s	○ 1.81s	✖ 1.23min	40x
i2c_k1	0 0.20s	○ 8.52s	○ 1.17s	1 ✔ 3.57s	Timeout	✔ 13.17s	✔ 41.10min	**187x**
sha3_w1	0 0.24s	○ 13.73s	○ 13.89s	○ 14.32s	○ 31.38s	○ 41.34s	✔ 1.19min	1x
sha3_r1	0 0.22s	Timeout	○ 0.36s	○ 0.34s	Timeout	Timeout	Timeout	964x
sha3_w2	0 0.24s	○ 0.36s	○ 0.64s	○ 22.49s	○ 34.80s	○ 21.78s	Timeout	2,644x
sha3_s1	0 0.20s	○ 3.27s	1 ✔ 0.33s	○ 10.15s	✔ 5.98s	✔ 3.77s	✖ 1.60min	25x
pairing_w1	0 18.49s	Timeout	○ 41.20s	○ 45.44s	Timeout	Timeout	Timeout	963x
pairing_k1	0 18.46s	Timeout	○ 41.79s	○ 42.03s	Timeout	Timeout	Timeout	963x
pairing_w2	0 18.42s	Timeout	○ 41.11s	○ 41.24s	Timeout	Timeout	Timeout	963x
reed_b1	0 0.33s	○ 1.79s	○ 1.09s	○ 1.81s	○ 5.44s	○ 5.52s	Timeout	10,428x
reed_o1	0 0.36s	○ 1.35s	○ 0.85s	○ 0.89s	○ 3.59s	○ 3.63s	✖ 9.50h	9,426x
sdram_w2	0 0.18s	2 ✔ 2.27s	○ 0.37s	○ 25.56s	Timeout	✔ 2.59s	Timeout	22,231x
sdram_k2	2 0.83s	Repaired by preprocessing			✔ 1.17s	✔ 1.20s	Timeout	48,157x
sdram_w1	0 0.18s	○ 0.32s	○ 0.38s	○ 0.62s	Timeout	○ 1.65s	✖ 6.91h	15,055x

them to the short names used throughout this paper. We created I/O traces from the provided ground truth versions of each circuit. We had to manually remove a tri-state bus and an asynchronous reset for two benchmarks as these constructs are not supported by RTL-Repair. This conversion could be automated in the future. The source code of RTL-Repair, our modified version of CirFix and all experimental scripts are available on GitHub: https://github.com/ekiwi/rtl-repair

6.2 Quality of Repairs

The most important metric for a repair tool is the number of bugs it can successfully repair. This requires us to classify any repair the tool comes up with as correct or incorrect. The authors of CirFix followed a two-step approach: (1) By design, all repairs that CirFix returns pass the provided testbench. These repairs were described to be "plausible". (2) In a second step, the first author of the paper would manually inspect each "plausible" repair and determine whether the repair is "correct". [1] We also inspected the repairs CirFix performed and found that many seemed incorrect to us.

Many of these disagreements relate to what each research team focuses on repairing. It appears that the CirFix authors are focused on repairing the Verilog simulation of a circuit, which CirFix accomplishes in many cases. However, the goal of RTL-Repair is to repair the circuit that is described by the Verilog simulation and not just the simulation itself. Under this framing, repairs that fix the simulation but lead to synthesis-simulation mismatch (see Section 2.1) are incorrect. Since these mismatches are notoriously difficult to debug, CirFix might cause more work than it saves.

[1]Source: personal communication with the authors.

A common way to detect synthesis-simulation mismatch is so-called gate-level simulation. For this purpose, we take the output of our synthesis tool in the form of a low-level Verilog description and plug it into the original testbench. Sometimes gate-level simulation fails, not because of an actual mismatch but because of various X-propagation issues. Therefore we only perform the gate-level simulation check if it works with the ground truth version of the circuit. We add another automated check for simulator compatibility: If the original circuit works with the open-source iverilog simulator [47], the repaired version should also work with iverilog. This helps us filter out repairs that rely on race conditions or otherwise ill-defined Verilog features.

The importance of avoiding synthesis-simulation mismatch is illustrated by the mux_w1 benchmark, which CirFix repairs through a "clever" combination of blocking and non-blocking assignments. A value is overwritten by a non-blocking statement, which appears in the program order *before* the blocking statement, which assigns the original default value. This repair fixes the simulation but is not correctly understood by the synthesis tool, leading to a much harder-to-detect and debug problem for the developer to deal with.

Finally, we noticed a problem with the testbench accompanying the decoder benchmarks. It does not adequately test all functionality of the design. We thus added an "extended" testbench that tests all relevant input combinations. This test shows one of the advantages of minimizing the number of repairs in the RTL-Repair algorithm: It ensures that RTL-Repair only changes code exercised by the testbench. CirFix, on the other hand, ends up destroying functional parts of the circuit that were not exercised by the testbench. The second decoder benchmark contains errors in parts of the design that were never tested by the original testbench and thus cannot be repaired by any tool. If we provide RTL-Repair with the extended testbench, it successfully finds the complete repair.

Overall, RTL-Repair finds 16 repairs that pass all our tests, while CirFix finds 10. Figure 8 features a qualitative comparison of four benchmarks, highlighting the strengths and weaknesses of both tools. RTL-Repair also provides only two incorrect repairs. The first one is due to the shortcomings in the decoder testbench. For the shift_k1 benchmark, RTL-Repair incorrectly determines that no repair is necessary since the synthesized circuit looks correct. This could easily be filtered out by running the original testbench once after a successful repair. With our more extensive testing in place, we notice that only two multi-edit repairs generated by CirFix are considered correct (counter_k1 and flop_w2). This calls into question CirFix's ability to generate multi-edit repairs that take full advantage of the genetic algorithm.

6.3 Repair Speed

Table 5 shows how long RTL-Repair and CirFix take for each repair. We used a timeout of 60 seconds for RTL-Repair and 16 hours for CirFix. RTL-Repair generally provides results in a small number of seconds, often several orders of magnitude faster than CirFix. It gives almost instant feedback allowing a user to quickly decide whether they want to use the repair suggestion.

We compare the adaptive windowing technique used by RTL-Repair to the basic synthesizer. For benchmarks with small testbench lengths, the basic synthesizer is faster. But for longer testbenches like the mux benchmarks, the adaptive windowing approach leads to faster results. It allows us to solve two more benchmarks compared to the basic synthesizer.

Under normal circumstances, RTL-Repair tries out repair templates in sequence and immediately returns as soon as a repair is found. The left half of table 5 shows what happens if we turn off this early exit. We can see that the repair templates do not overlap; only a single repair template per benchmark generates a repair. Each repair template fixes between three and four of the benchmarks, demonstrating that the templates are not specific to a single bug. We can also see that the number of changes for each repair is small. Most often, only one or two changes are enough; the maximum is three changes to generate a correct repair. Five benchmarks are fixed directly by our static-analysis-based preprocessing phase, demonstrating the importance of combining static analysis with more sophisticated repair techniques.

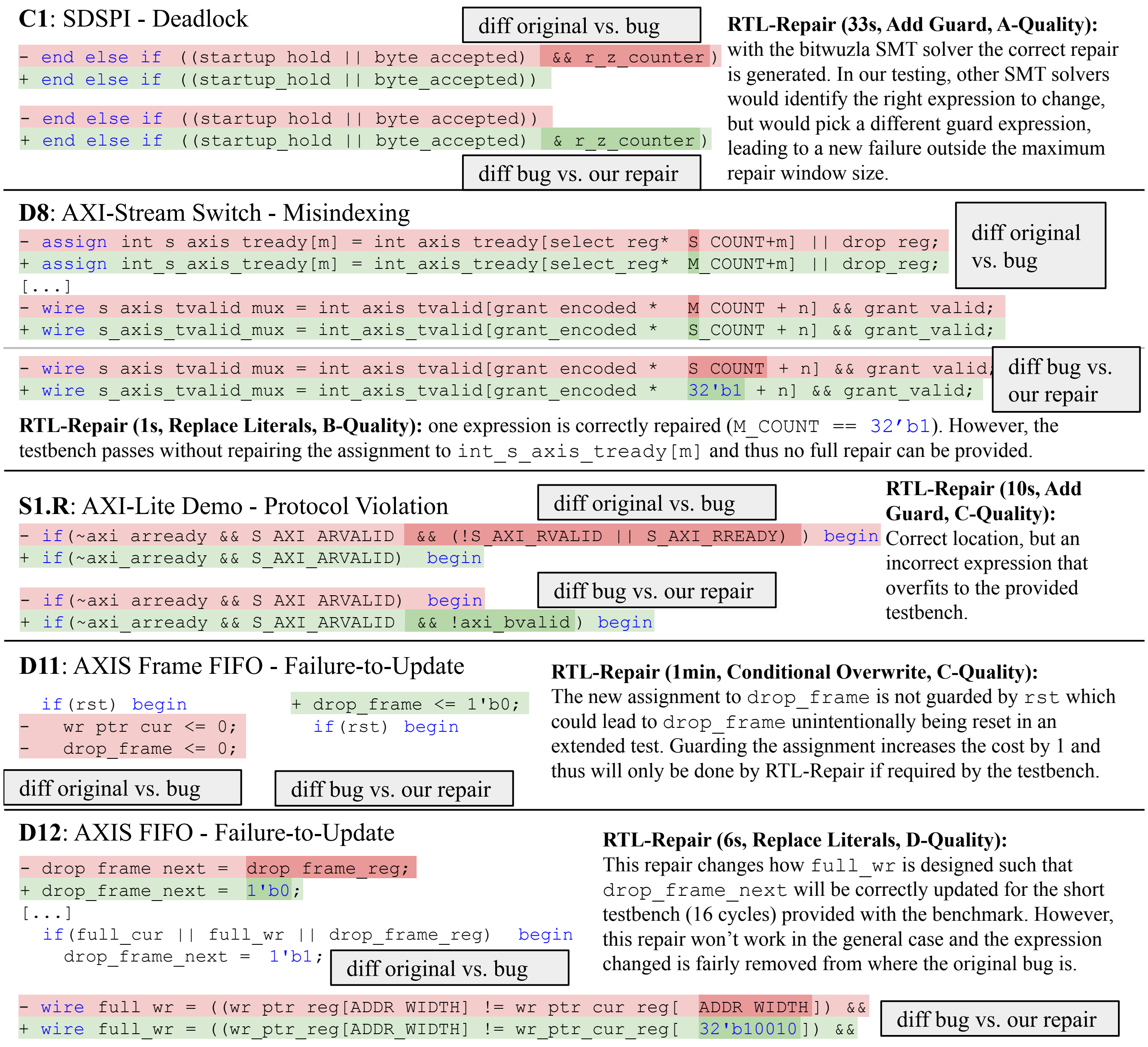
6.4 Open-Source Bug Repair

In addition to the CirFix benchmarks, which were specifically created to test automated repair tools, we also applied our RTL-Repair tool to a set of bugs mined from git commits to open-source FPGA hardware projects [31]. Of the 20 reproducible bugs provided by the prior work [31], we are able to use 12 with RTL-Repair. The other 8 contain non-synthesizable Verilog, use SystemVerilog features that our parser is not able to deal with, or lack a ground-truth repair.

Table 6 shows our results. Overall, RTL-Repair provides repairs that pass the provided testbench for 9 out of 12 bugs. However, since this set of bugs was never intended to be used as a benchmark for automated repair, most testbenches are quite minimalistic and only enough to demonstrate the bug. We thus manually inspect each repair and rate it on the following scale: (A) repair matches the

Table 6: RTL-Repair results for bugs from open-source projects collected by the authors of "Debugging in the Brave New World of Reconfigurable Hardware" (Table 2 in [31]). All results were obtained using the incremental synthesizer and with a timeout of 2min. "Bug Diff" indicates how many lines need to be added or removed in order to go from the repaired to the buggy version of the circuit. "TB" shows the number of steps in the provided testbench.

	Bug Diff	TB	Result and Quality			Template
D4	+27 / -26	185	Timeout			
D8	+2 / -2	14	**1** ✔	1.06s	B	Replace Literals
D9	+2 / -2	523k	Timeout			
D11	+0 / -2	17	**1** ✔	54.12s	C	Cond. Overwrite
D12	+1 / -1	16	**1** ✔	6.06s	D	Replace Literals
D13	+1 / -3	6	**3** ✔	1.54s	C	Cond. Overwrite
C1	+1 / -1	523k	**1** ✔	33.17s	A	Add Guard
C3	+1 / -7	523k	○	21.56s		
C4	+1 / -1	10	**1** ✔	1.83s	A	Add Guard
S1.R	+1 / -1	10	**1** ✔	10.23s	C	Add Guard
S1.B	+2 / -2	10	**2** ✔	9.09s	D	Add Guard
S2	+1 / -2	45	**1** ✔	0.73s	C	Replace Literals
S3	+12 / -35	13	**2** ✔	6.89s	D	Replace Literals



C1: SDSPI - Deadlock

diff original vs. bug

```
- end else if ((startup_hold || byte_accepted) && r_z_counter)
+ end else if ((startup_hold || byte_accepted))
```

```
- end else if ((startup_hold || byte_accepted))
+ end else if ((startup_hold || byte_accepted) & r_z_counter)
```

diff bug vs. our repair

RTL-Repair (33s, Add Guard, A-Quality): with the bitwuzla SMT solver the correct repair is generated. In our testing, other SMT solvers would identify the right expression to change, but would pick a different guard expression, leading to a new failure outside the maximum repair window size.

D8: AXI-Stream Switch - Misindexing

diff original vs. bug

```
- assign int_s_axis_tready[m] = int_axis_tready[select_reg* S_COUNT+m] || drop_reg;
+ assign int_s_axis_tready[m] = int_axis_tready[select_reg* M_COUNT+m] || drop_reg;
[...]
- wire s_axis_tvalid_mux = int_axis_tvalid[grant_encoded * M_COUNT + n] && grant_valid;
+ wire s_axis_tvalid_mux = int_axis_tvalid[grant_encoded * S_COUNT + n] && grant_valid;
```

```
- wire s_axis_tvalid_mux = int_axis_tvalid[grant_encoded * S_COUNT + n] && grant_valid;
+ wire s_axis_tvalid_mux = int_axis_tvalid[grant_encoded * 32'b1 + n] && grant_valid;
```

diff bug vs. our repair

RTL-Repair (1s, Replace Literals, B-Quality): one expression is correctly repaired (`M_COUNT == 32'b1`). However, the testbench passes without repairing the assignment to `int_s_axis_tready[m]` and thus no full repair can be provided.

S1.R: AXI-Lite Demo - Protocol Violation

diff original vs. bug

```
- if(~axi_arready && S_AXI_ARVALID && (!S_AXI_RVALID || S_AXI_RREADY) ) begin
+ if(~axi_arready && S_AXI_ARVALID) begin
```

```
- if(~axi_arready && S_AXI_ARVALID) begin
+ if(~axi_arready && S_AXI_ARVALID && !axi_bvalid) begin
```

diff bug vs. our repair

RTL-Repair (10s, Add Guard, C-Quality): Correct location, but an incorrect expression that overfits to the provided testbench.

D11: AXIS Frame FIFO - Failure-to-Update

```
  if(rst) begin
-   wr_ptr_cur <= 0;
-   drop_frame <= 0;
```

diff original vs. bug

```
+ drop_frame <= 1'b0;
  if(rst) begin
```

diff bug vs. our repair

RTL-Repair (1min, Conditional Overwrite, C-Quality): The new assignment to `drop_frame` is not guarded by `rst` which could lead to `drop_frame` unintentionally being reset in an extended test. Guarding the assignment increases the cost by 1 and thus will only be done by RTL-Repair if required by the testbench.

D12: AXIS FIFO - Failure-to-Update

```
- drop_frame_next = drop_frame_reg;
+ drop_frame_next = 1'b0;
[...]
  if(full_cur || full_wr || drop_frame_reg) begin
    drop_frame_next = 1'b1;
```

diff original vs. bug

RTL-Repair (6s, Replace Literals, D-Quality): This repair changes how `full_wr` is designed such that `drop_frame_next` will be correctly updated for the short testbench (16 cycles) provided with the benchmark. However, this repair won't work in the general case and the expression changed is fairly removed from where the original bug is.

```
- wire full_wr = ((wr_ptr_reg[ADDR_WIDTH] != wr_ptr_cur_reg[ ADDR_WIDTH]) &&
+ wire full_wr = ((wr_ptr_reg[ADDR_WIDTH] != wr_ptr_cur_reg[ 32'b10010]) &&
```

diff bug vs. our repair

Figure 9: Repairs produced by RTL-Repair for the Open-Source bugs discussed in Section 6.4.

ground truth exactly, (B) repair performs some of the changes from the ground truth, (C) repair changes the same expression as the ground truth but in a different way, and (D) change is very different from the ground truth. Figure 9 shows several example repairs.

In order to be able to tackle his new challenging benchmarkset, we needed to improve our repair templates in order to make them more powerful. The Add Guard template, for example, used to only allow inversion of boolean conditions. We added the ability to add another boolean condition as a guard. While we had to improve our templates, we were still able to implement them in under 150 lines of Python each, and we were able to keep the number of templates at three.

While we did improve our templates, the core synthesis algorithm remained largely untouched. This shows that while templates need to be carefully engineered to work across a large set of repair scenarios, the basic synthesis technique proposed in this paper can be applied to a wide range of designs. Note that none of these more realistic benchmarks struggled with synthesis-simulation mismatch, and none were repaired by preprocessing alone. However, while the bugs are all mined from open-source projects, most only come with artificially short testbenches that are provided only to demonstrate each bug. This leads to many possible repairs that can make the testbench pass. While RTL-Repair always provides a very small repair, some of them are not very good. Sampling multiple repairs and presenting them to the user could be a future fix to this problem.

7 RELATED WORK

RTL-Repair and CirFix are currently the only end-to-end tools that generate complete repairs from a buggy Verilog source code and testbench alone. Recent work using LLMs assumes the precise fault location is known [5]. There are many fault localization approaches for hardware, but none of them are exact [24, 40, 49]. Some other repair tools rely on C reference models [7] or formal LTL properties [13, 16] instead of testbenches.

There has been work in the past on symbolic-analysis-based repair for hardware [13, 15, 32]. However, it appears that none of these approaches can deal with long-running testbenches and instead focus on bugs that appear after one or two cycles of execution. The work by *Chang et.al.* [15] is noteworthy because it uses a two-step approach that first identifies faulty expressions and then synthesizes a repair to replace them. A similar approach was independently discovered years later for software repair with the Angelix tool [14, 34, 36].

8 DISCUSSION

RTL-Repair clearly illustrates the power of symbolic analysis-based repair techniques, providing more correct repairs orders of magnitude faster than the generate-and-validate based CirFix tool. We carefully designed RTL-Repair to work with the exact same assumptions as the prior work to make it a drop-in replacement for CirFix. This shows that symbolic repair does not require formal specifications.

Our repair templates are directly applied to the Verilog AST, making it trivial to map the repair suggested by the synthesizer back to the original design. Initially, we explored templates that worked on the transition system representation, which led to repairs that proved difficult to automatically incorporate into the high-level Verilog code. Because of our standardized interface to the repair synthesizer, new templates are easy to add.

We introduce gate-level simulation as a new standard for evaluating automated repairs of hardware designs. This ensures that the users of these tools are not in for a bad surprise when the automated repair makes their Verilog simulation work but then leads to silent bugs in the actual circuit when it is mapped to an FPGA or taped out in a VLSI process.

RTL-Repair provides repair suggestions in a matter of seconds. Through our adaptive windowing technique, this remains true, even for larger benchmarks. With this level of responsiveness, we imagine that RTL-Repair could be integrated into a Verilog IDE to directly provide quick repair suggestions, similar to tools like GitHub Copilot [21]. This would require more research into how exactly RTL-Repair could be integrated with various forms of testbenches and formal tests.

ACKNOWLEDGMENT

We would like to thank our shepherd, Aditya Kanade, and the anonymous reviewers for helping us improve our paper. We are also grateful to the CirFix authors, in particular Hammad Ahmad and Westley Weimer, for detailed answers to our questions and for providing an easy-to-use artifact with their paper.

This work was supported in part by Semiconductor Research Corporation, by NSF grants CCF-1900968, CCF-1908870, CCRI-2016662, POSE-2303735 and by SLICE Lab industrial sponsors and affiliates Amazon, AMD, Apple, Google, Intel, NVIDIA and Qualcomm, as well as by SKY lab industrial sponsors and affiliates Accenture, AMD, Anyscale, Google, IBM, Intel, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, SAP, Uber, and VMware. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact includes the full implementation of our RTL-Repair tool, an improved version of the CirFix tool, which we used in our comparison (see Section 6.1), as well as all scripts and benchmarks used in our evaluation. We include scripts to collect all necessary data to recreate Tables 1, 2, 4, 5 and 6. We also include a small demo to demonstrate the reusability of our tool.

A.2 Artifact check-list (meta-information)

- **Run-time environment:**
  - Synopsys VCS Simulator
  - Python 3.10
  - Rust 1.76.0
  - OSS CAD Suite 2022-06-22:
    - Verilator 4.x (tested with Verilator 4.225)
    - bitwuzla 1.0-prerelease
    - Icarus Verilog version 12.0
    - Yosys 0.18+29
- **Experiments:** OSDD calculation, CirFix evaluation, RTL-Repair evaluation with different settings, correctness checks
- **How much disk space required (approximately)?:** 20GB
- **How much time is needed to prepare workflow (approximately)?:** 10min (assuming VCS, Python and Rust are already available)
- **How much time is needed to complete experiments (approximately)?:** 26h (2h + 24h for full CirFix evaluation)
- **Publicly available?:** Yes. On Github: https://github.com/ekiwi/rtl-repair
- **Code licenses:** BSD 3-Clause License
- **Archived DOI:** https://doi.org/10.5281/zenodo.10798649

A.3 Description

A.3.1 How to access. We recommend cloning the GitHub repository for the latest code:

https://github.com/ekiwi/rtl-repair

A.3.2 Software dependencies. Our artifact has been tested on Ubuntu 20.04.6 LTS with VCS 2021.09, Python 3.10.6, and Rust 1.76.0.

A.4 Installation

You have to install all software mentioned in the artifact checklist. Please see the Readme.md provided with the artifact for more detailed information.

A.5 Evaluation and expected results

The artifact contains scripts to reproduce the following tables:

- Performance Overview (Table 1)
- OSDD (Table 2)
- Repair Correctness (Table 4)
- Repair Speed (Table 5)
- Open-Source Bug Results (Table 6)

Detailed instructions are provided in the Readme.md included with the artifact.

A.6 Experiment customization

We provide a demo with the artifact, which makes it easy to introduce new bugs into a Verilog design, run a test, and use RTL-Repair to get a repair suggestion. Details are provided in the Readme.md included with the artifact.

To experiment with different repair templates, please have a look at the files in rtlrepair/templates.

A.7 Methodology

Submission, reviewing, and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] IEC/IEEE International Standard - Verilog(R) Register Transfer Level Synthesis. *IEEE/IEC 62142*, 2005.
[2] IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language. *IEEE Std. 1800*, 2017.
[3] IEEE Standard for VHDL Language Reference Manual. *IEEE Std. 1076*, 2019.
[4] IEEE Standard for Universal Verification Methodology Language Reference Manual. *IEEE Std. 1800.2*, 2020.
[5] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. Fixing Hardware Security Bugs with Large Language Models. *arXiv preprint arXiv:2302.01215*, 2023.
[6] Hammad Ahmad, Yu Huang, and Westley 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*, pages 990–1003, 2022.
[7] Bijan Alizadeh and Masoud Shiroei. Automatic Correction of RTL Designs Using a Lightweight Partial High Level Synthesis. *Integration*, 91:173–181, 2023.
[8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. In *DAC Design Automation Conference 2012*, 2012.
[9] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.
[10] Clark Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*. 2008.
[11] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu, et al. Bounded Model Checking. *Advances in computers*, 58(11):117–148, 2003.
[12] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. *νz*-an optimizing smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21*, pages 194–199. Springer, 2015.
[13] Roderick Bloem and Franz Wotawa. Verification and Fault Localization in VHDL Programs. *Journal of the Telematics Engineering Society (TIV)*, 2:30–33, 2002.
[14] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic Debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 121–130, 2011.
[15] Kai-hui Chang, Ilya Wagner, Valeria Bertacco, and Igor L Markov. Automatic Error Diagnosis and Correction for RTL Designs. In *2007 IEEE International High Level Design Validation and Test Workshop*, pages 65–72. IEEE, 2007.
[16] Matthias Cosler, Frederik Schmitt, Christopher Hahn, and Bernd Finkbeiner. Iterative Circuit Repair Against Formal Specifications. *arXiv preprint arXiv:2303.01158*, 2023.
[17] Clifford E Cummings et al. Nonblocking assignments in verilog synthesis, coding styles that kill! *SNUG (Synopsys Users Group) 2000 User Papers*, 2000.
[18] Amelia Dobis, Kevin Laeufer, Hans Jakob Damsgaard, Tjark Petersen, Kasper Juul Hesse Rasmussen, Enrico Tolotto, Simon Thye Andersen, Richard Lin, and Martin Schoeberl. Verification of Chisel Hardware Designs with ChiselVerify. *Microprocessors and Microsystems*, 96:104737, 2023.
[19] Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon Davidmann. Verilog HDL and Its Ancestors and Descendants. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–90, 2020.
[20] Tristan Gingold et al. GHDL - VHDL 2008/93/87 simulator. http://ghdl.free.fr/, 2023.
[21] GitHub. GitHub Copilot. https://copilot.github.com/, 2023.
[22] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated Program Repair. *Communications of the ACM*, 62(12):56–65, 2019.
[23] Chris Higgs, Stuart Hodgson, and Eric Wieser. cocotb. https://github.com/cocotb/cocotb, 2021.
[24] Tai-Ying Jiang, C-NJ Liu, and Jing Ya Jou. Estimating Likelihood of Correctness for Error Candidates to Assist Debugging Faulty HDL Designs. In *2005 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 5682–5685. IEEE, 2005.
[25] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*.
[26] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K Ganai. Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.
[27] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.
[28] Richard Lin and Kevin Laeufer. ChiselTest. https://github.com/ucb-bar/chiseltest, 2024.
[29] Fan Long and Martin Rinard. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312, 2016.
[30] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, and Baris Kasikci. Debugging in the Brave New World of Reconfigurable Hardware. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 946–962, 2022.
[31] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, and Baris Kasikci. Debugging in the Brave New World of Reconfigurable Hardware. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.
[32] Jean Christophe Madre, Olivier Coudert, and Jean Paul Billon. Automating the Diagnosis and the Rectification of Design Errors with PRIAM. In *The Best of ICCAD: 20 Years of Excellence in Computer-Aided Design*, pages 17–27. Springer, 1989.
[33] Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*. 2021.
[34] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701, 2016.
[35] Don Mills and Clifford E Cummings. RTL Coding Styles That Yield Simulation and Synthesis Mismatches. In *SNUG (Synopsys Users Group) 1999 Proceedings*, 1999.
[36] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program Repair via Semantic Analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.
[37] Aina Niemetz and Mathias Preiner. Bitwuzla at the SMT-COMP 2020. *CoRR*, abs/2006.01621, 2020.
[38] Aina Niemetz, Mathias Preiner, Claire Wolf, and Armin Biere. Btor2, BtorMC and Boolector 3.0. In *International Conference on Computer Aided Verification*, pages 587–595. Springer, 2018.
[39] Zichao Qi, Fan Long, Sara Achour, and Martin 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*, pages 24–36, 2015.

[40] Jiann-Chyi Ran, Yi-Yuan Chang, and Chia-Hung Lin. An Efficient Mechanism for Debugging RTL Description. In *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, 2003. Proceedings.*, pages 370–373. IEEE, 2003.

[41] Wilson Snyder et al. Verilator. https://www.veripool.org/wiki/verilator, 2023.

[42] Stuart Sutherland and Don Mills. Synthesizing systemverilog busting the myth that systemverilog is only for verification. *SNUG Silicon Valley*, page 24, 2013.

[43] Synopsys. VCS. https://www.synopsys.com/verification/simulation.html, 2023.

[44] Shinya Takamaeda-Yamazaki. Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL. In *Applied Reconfigurable Computing: 11th International Symposium, ARC 2015, Bochum, Germany, April 13-17, 2015, Proceedings 11*, pages 451–460. Springer, 2015.

[45] Lenny Truong, Steven Herbst, Rajsekhar Setaluri, Makai Mann, Ross Daly, Keyi Zhang, Caleb Donovick, Daniel Stanley, Mark Horowitz, Clark Barrett, et al. fault: A Python Embedded Domain-Specific Language for Metaprogramming Portable Hardware Verification Components. In *International Conference on Computer Aided Verification*, CAV'20, 2020.

[46] Mike Turpin. The dangers of living with an x (bugs hidden in your verilog). In *Synopsys Users Group Meeting*, 2003.

[47] Stephen Williams et al. Icarus Verilog. https://steveicarus.github.io/iverilog/, 2023.

[48] Claire Wolf and Johann Glaser. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[49] Jiang Wu, Zhuo Zhang, Deheng Yang, Xiankai Meng, Jiayu He, Xiaoguang Mao, and Yan Lei. Fault Localization for Hardware Design Code with Time-Aware Program Spectrum. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 537–544. IEEE, 2022.

[50] Keyi Zhang, Zain Asgar, and Mark Horowitz. Bringing Source-Level Debugging Frameworks to Hardware Generators. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, 2022.