Feedback-Guided Circuit Structure Mutation for Testing Hardware Model Checkers

Chengyu Zhang, Minquan Sun, Jianwen Li, Ting Su, Geguang Pu†
Software Engineering Institute, *East China Normal University*
Shanghai, China
dale.chengyu.zhang@gmail.com, sunminquan2016@163.com, {jwli, tsu, ggpu}@sei.ecnu.edu.cn

***Abstract*—We introduce *Circuit Structure Mutation*, a simple but effective mutation-based testing approach, for testing hardware model checkers. The key idea is to mutate the existing And-Inverter Graph (AIG) circuit by manipulating the relations among the components in the graph while preserving the validity of the mutant. Based on *Circuit Structure Mutation*, we implemented a feedback-guided testing tool named `Hammer`. In our evaluation, `Hammer` shows its effectiveness on finding bugs, increasing test coverage, and finding performance optimization chances, which can help the hardware model checker developers improve the reliability and the performance of their tools.**

***Index Terms*—Hardware Model Checking, Software Testing, Test Case Generation**

I. Introduction

Model checking is a well-known technique for verifying the correctness of the software and hardware systems [1], [2]. Given a property P and a model M, model checking is to check whether P holds on M. If P is violated on M, a counterexample can be given to evidence the property violation. Hardware model checkers are the tools particularly for verifying hardware designs automatically. Along with the recent significant progress in this field, hardware model checkers have been widely used in both industry and academia [3], [4], [5], [6].

To encourage the improvement of hardware model checkers, Hardware Model Checking Competition (HWMCC) [7] was established. The HWMCC benchmarks consist of realistic hardware designs from both industry and academia. The performance and the reliability of the participant hardware model checkers are evaluated on these benchmarks. The HWMCC benchmarks also become the test suite for testing the checkers while developing the hardware model checkers. However, since it is difficult to collect new benchmarks, the developers of hardware model checkers are not able to validate and improve the tools beyond the HWMCC benchmarks.

In fact, the developers of hardware model checkers suffer from reliability issues and performance regressions during the development. For example, unsoundness issue is a kind of reliability issue in hardware model checkers. It makes checkers produce incorrect checking results and threats the safety of the hardware designs under verification. Runtime crash is another kind of reliability issue. It obstructs the checking process and causes that the hardware designed cannot be verified. The performance regressions may appear when the developer adds optimizations or modifies code in the checker. These updates could make the new version significantly slower than the old version on some cases. Due to the limitation of the existing benchmarks, these issues and regressions cannot be quickly addressed before using in wild. Hence, an automated tool that can find out the reliability issues and performance regressions will be helpful for the hardware model checker development.

In this paper, we introduce a simple but effective mutation-based testing approach called *Circuit Structure Mutation*, to automatically uncover reliability issues and performance regressions in the hardware model checkers. The key idea of this approach is to manipulate the relations among the components in the And-Inverter Graph (AIG) while preserving the validity of the mutant. The traditional mutation-based testing approaches cannot be applied for testing hardware model checkers, since they will generate the mutants that correspond to invalid hardware designs that will be rejected by the hardware model checkers at early stage. Besides, we enhance *Circuit Structure Mutation* with feedback-guided strategy to make it increase test coverage and find performance optimization chances more efficiently in practice.

Based on *Circuit Structure Mutation*, we implement a feedback-guided testing tool called `Hammer` and evaluate `Hammer` on testing three state-of-the-art hardware model checkers ABC [8], nuXmv [9] and AVY [10]. The results show that `Hammer` is promising. `Hammer` find 10 unique issues in these hardware model checkers and can significantly increase the test coverage and effectively find potential optimization chances. The performance of `Hammer` is significantly better than the only existing hardware model checker testing tool `aigfuzz`. In this sense, `Hammer` provides a novel way for testing hardware model checkers to complement `aigfuzz`.

In summary, we make the following contributions:

- We propose *Circuit Structure Mutation*, a simple but effective approach to generate test inputs and benchmarks for hardware model checkers.
- We enhance *Circuit Structure Mutation* with feedback-guided strategy to increase test coverage and find more performance optimization chances.
- We implemented `Hammer`, a tool for testing hardware model checkers, based on *Circuit Structure Mutation*,
- In the evaluation, we found 10 unique bugs in three hardware model checkers using `Hammer` and show that the feedback-guided strategy works for enhancement.

†Geguang Pu is the corresponding author.

This paper is organized as follows: Section II introduces the related work. Section III illustrates *Circuit Structure Mutation* with an example. Section IV introduces the definition of this approach, describes the feedback-guided mutation algorithm and the implementation of `Hammer`. Section V shows the results of the empirical evaluation on `Hammer` and compares `Hammer` with `aigfuzz`. Section VI concludes this paper.

II. Related work

Mutation-based testing. Mutation-based testing is an automatic software testing technique for discovering software vulnerabilities by randomly mutating existing test inputs to generate a large amount of new test inputs [11]. Such test inputs could be malformation, so that they could trigger the corner cases in the software being tested. Coverage-guided testing is an enhanced testing technique that uses the code coverage information to generate more diverse test inputs for covering more program statements and paths. `AFL` [12] is a representative coverage-guided mutation-based testing tool. The basic idea of `AFL` is to mutate the seed files to generate new test inputs, and then incorporate the mutants that cover new paths into the seed set for the following testing process. `AFL` has shown its effectiveness in finding security-oriented vulnerabilities in important open-source software, such as Mozilla Firefox [13], MySQL [14], and OpenSSL [15]. However, `AFL` only works with bit, word, and token level mutations. There is a group of following work related to `AFL`. For example, FairFuzz [16] and Steelix [17] improve the performance of `AFL` on the binary level. However, for the software that requires structured inputs, traditional mutation-based techniques usually generated syntactically invalid test inputs that will be rejected by the software. Thus, several syntax-aware mutation-based techniques were proposed for generating syntactically valid inputs [18], [19], [20]. However, syntax-aware mutation-based techniques still do not work for testing hardware model checkers, since AIG has to be not only syntax-valid but also reasonable for being a hardware design. Therefore, a tailored semantic-aware mutation is required for testing hardware model checkers.

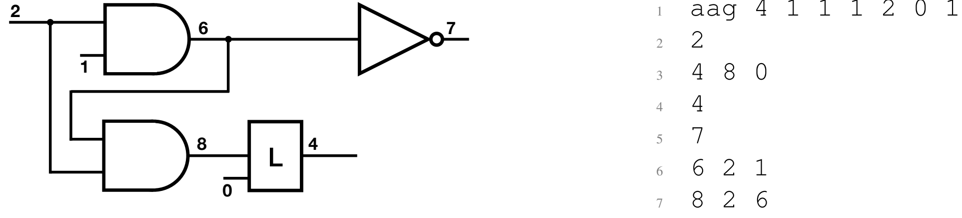
Testing verification tools. Verification tools are usually for verifying the correctness of models, software, and hardware. However, the verification tools themselves should also be validated. Recent work targets many different kinds of verification tools. Zhang *et al.* [21] and Klinger *et al.* [22] proposed reachability query and check synthesis approaches respectively to generate test inputs for software model checkers and found many bugs in them. FuzzSAT, CNFuzz, and 3SAT [23] are fuzzers for SAT solvers, they found 14 bugs in 7 SAT solvers via differential testing. SMT solvers are recently being intensively tested. Many SMT testing tools have been proposed, such as YinYang [24], [25], STORM [26], StringFuzz [27], and BanditFuzz [28]. Despite the reliability of verification tools that have been noticed, it still lacks tools and approaches for testing hardware model checkers. The tool `aigfuzz` in AIGER library [29] is the only existing automated testing tool for hardware model checkers which is based on a purely random generation strategy. However, due to the large generation space, the random generation strategy can rarely generate the test inputs that explore deeper into the code. Thus, the mutation-based strategies would be promising, while there are no mutation-based tools for hardware model checkers yet.

III. Illustrative example

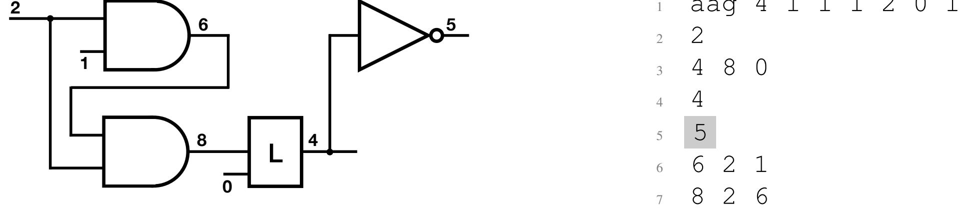
To address the above challenges, we propose *Circuit Structure Mutation* for testing hardware model checkers. It can also be adapted to test the software that uses AIG as input, such as Electronic Design Automation (EDA) and electronic simulation software. In this section, we first introduce the background of hardware model checker and then give an illustrative example of our approach.

Background. And-Inverter Graph (AIG) is a directed, acyclic graph for describing a logic circuit. A typical AIG only contains two kinds of components: *And* gate and *Inverter* gate. An arbitrary combinatorial logic circuit can be represented by AIG. To support sequential logic, *Latch* gate is added to AIG, which is called sequential AIG. Figure 1a shows an example of AIG. AIGER format is proposed to provide a unified input format of AIG for hardware model checkers. The right side of Figure 1a presents an ASCII version AIGER file corresponding to the circuit on the left side of Figure 1a. The first line of AIGER file specifies the numbers of literal (the identifier of the component) and each kind of component. Term `"aag"` denotes this AIGER file is written in ASCII version. The following numbers in the first line represent the numbers of components, inputs, latches, outputs, *And* gates, bad states, and invariant constraints in sequence. The following lines specify the relations among the components in sequence. For example, on the right side of Figure 1a, the second line says the input is assigned to literal 2. The third line says the latch is initialized to a constant 0, the next state of this latch is defined by the literal 8, and the output of this latch is assigned to literal 4. The following two lines say that the output of the circuit is literal 4 and the invariant constraint is defined by the literal 6 with a *Inverter* gate (represented by literal 7) respectively. The last two lines represent two *And* gates. One *And* gate uses literal 2 and constant 1 as the inputs and sets the output to literal 6, another one uses literal 2 and literal 6 as inputs and sets the output to literal 8.

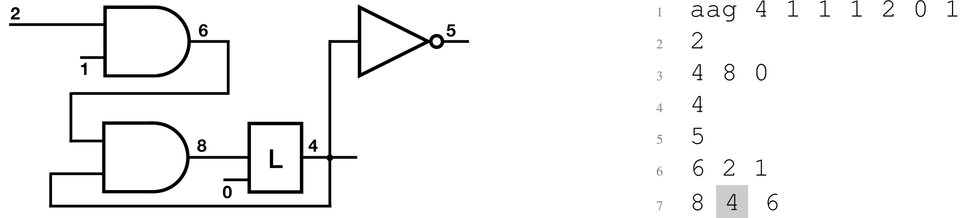
Illustrative example. The intuitive idea of *Circuit Structure Mutation* is to manipulate the relations among the components in AIGs. Figure 1b and Figure 1c give an intuitive example of how *Circuit Structure Mutation* works for the circuit in Figure 1a. First, *Circuit Structure Mutation* randomly selects a component in Figure 1a, *e.g.*, the invariant constraint with literal 7 (in line 5), then randomly mutates literal 7 to literal 5, which produces Figure 1b. Then, we continue *Circuit Structure Mutation* on the mutant of Figure 1b. An And-gate with literal 8 is selected (in line 7), and one of its inputs is mutated from literal 2 to literal 4, which produces Figure 1c. Note that the mutants generated by *Circuit Structure Mutation* are still syntax-valid and correspond to correct hardware designs,



(a) Example of AIG.



(b) Circuit structure mutant of Fig. 1a.



(c) Circuit structure mutant of Fig. 1b.

Fig. 1: Illustrative examples of *Circuit Structure Mutation*. Left side presents the AIGs in visual graph, right side presents the AIGs in AIGER format.

therefore, they can be used as the test inputs for the hardware model checkers. The test inputs can uncover crashes and explore unsoundness issues by cross-checking the results of two different hardware model checker implementations, which is called differential testing.

In fact, the mutant in Figure 1c triggered an unsoundness bug in AVY. We found this bug by comparing the results of two hardware model checker implementations ABC and AVY. For this mutant, ABC reports checking successfully (safe), while AVY reports a counterexample (unsafe). After checking the counterexample, it was identified as an issue in AVY and has been reported to the developers by us. This strategy is called differential testing which is used to uncover unsoundness bugs in this paper. Interestingly, if we remove the invariant constraint of the mutant in Figure 1c, it triggers an assertion failure in AVY. In Section V, we will shows more hardware model checker bugs found in our evaluation.

IV. Approach and Implementation

In this section, we define *Circuit Structure Mutation* on AIG. Then, introduce the feedback-guided strategy to enhance *Circuit Structure Mutation*. Finally, describe the implementation of hardware model checker testing framework `Hammer`.

A. Circuit Structure Mutation

We first give the definition of *Circuit Structure Mutation*. Given an AIG model G, which can be defined by a 5-tuple $\langle I, O, A, L, R_{real} \rangle$, where:

- I is the set of *Input*s in G.
- O is the set of *Output*s in G.
- A is the set of *And* gates in G.
- L is the set of *Latch* gates in G.
- R_{real} is the set of relations among the component in G.

Relation set R_{in} donates all possible relations of connecting the inputs with other components:

$$R_{in} = (I \times O) \cup (I \times A) \cup (I \times L)$$

Relation set R_{mid} donates all possible relations among gates:

$$R_{mid} = (L \times A) \cup (A \times L) \cup (A \times A) \cup (L \times L)$$

Relation set R_{out} donates all possible relations of connecting other components with outputs:

$$R_{out} = (I \times O) \cup (A \times O) \cup (L \times O)$$

Relation set R_{real} of G satisfies the statement:

$$R_{real} \subseteq R_{in} \cup R_{mid} \cup R_{out}$$

Then, we define *Circuit Structure Mutation*.

Algorithm 1: *Circuit Structure Mutation*

```
  Input: AIGER file G
  Output: Mutant G'
1 Procedure Mutate(G):
2   Comp, i_max ← Parse(G)
3   while iterations > 0 do
4     comp ← random.choice(Comp)
5     I ← range(0, i_max)
6     i ← random.choice(I)
7     comp' ← comp.setInput(i)
8     Comp' ← (Comp − {comp}) ∪ {comp'}
9     if not Comp'.hasCirDep() then
10      Comp ← Comp'
11      iterations ← iterations − 1
12  G' ← buildAIGwith(Comp)
13  return G'
```

*Definition 1 (*Circuit Structure Mutation*):* Given an AIG model $G = \langle I, O, A, L, R_{real} \rangle$, arbitrary relation $(c_x, c_y) \in R_{real}$, and relation

$$r \in \{(c_z, c_y) | (c_z, c_y) \in R_{in} \cup R_{mid} \cup R_{out}\}$$

We construct a relation set R'_{real} by

$$R'_{real} = (R_{real} - \{(c_x, c_y)\}) \cup \{r\}$$

We say $G' = \langle I, O, A, L, R'_{real} \rangle$ is a circuit structure mutant. The transforming process from G to G' is called the circuit structure mutation. Then, we give an example using Fig. 1.

*Example 1 (*Circuit Structure Mutation*):* Given the AIG model G in Fig. 1a, the elements in the 5-tuple of G are $I = \{i_2\}$, $O = \{o_4, o_7\}$, $A = \{a_6, a_8\}$, $L = \{l_4\}$, and $R_{real} = \{(a_8, l_4), (l_4, o_4), (a_6, o_7), (i_2, a_6), (1, a_6), (i_2, a_8), (a_6, a_8)\}$ respectively. We take an arbitrary relation in R_{real}, for example, (a_6, o_7) and an arbitrary relation in $R_{out} = \{(i_2, o_4), (i_2, o_7)(l_4, o_4), (l_4, o_7), (a_6, o_4), (a_6, o_7), (a_8, o_4), (a_8, o_7)\}$, for example, (l_4, o_7). Then, we substitute (a_6, o_7) with (l_4, o_7) in R_{real}, which is

$$R'_{real} = (R_{real} - \{(a_6, o_7)\}) \cup \{(l_4, o_7)\}$$

Now, $R'_{real} = \{(a_8, l_4), (l_4, o_4), (l_4, o_7), (i_2, a_6), (1, a_6), (i_2, a_8), (a_6, a_8)\}$. Next, we use R'_{real} to construct a new circuit structure mutant $G' = \langle I, O, A, L, R'_{real} \rangle$. The mutant G' is the AIG model presented in Fig. 1b. *Circuit Structure Mutation* can further be applied on the AIG in Fig. 1b to generate the AIG in Fig. 1c. The mutation process is similar.

In practice, we could also consider randomly add or remove inverters when doing *Circuit Structure Mutation*. For example, given a relation (c_a, c_b), we could add a inverter between c_a and c_b, when constructing R'_{real} using this relation.

Algorithm 1 shows the process of the *Circuit Structure Mutation* in our implementation. The input of Algorithm 1 is an AIGER file G. First, G is parsed to obtain the set of components $Comp$ and the maximum literal index i_{max} (line 2). Then, the main mutation process will be iterated multiple times according to the user-configured value *iterations* (line 3). In each iteration, a component *comp* is randomly selected from the set (line 4) and a random literal index i is selected in the range from 0 to the maximum literal index (line 5). The random literal index i represents the (inverted) output components in $Comp$ or a constant. By assigning one input of the component *comp* to the random literal index i, a new component object $comp'$ is generated (line 7). The old component object *comp* is substituted with the new component object $comp'$ in the component set $Comp$ (line 8 and 10). After the iterations, a new AIGER file G' constructed by the updated component object set $Comp$ is returned by the procedure (line 12). In practice, in case $comp'$ may cause the invalid circular dependency of *And* gates, a circular dependency detector is added after obtaining $comp'$. If a circular dependency is detected, $comp'$ will be discarded (line 9). Given a group of seed files, `Hammer` without feedback-guided strategy will execute Algorithm 1 for each file in the group to generate new test inputs. We next introduce how to enhance *Circuit Structure Mutation* with feedback-guided strategy.

Algorithm 2: Feedback-guided strategy process

```
  Input: Seed files F, Checker c
  Output: Test suites T
1 Procedure FeedbackMutation(c, F):
2   while not timeout do
3     forall f ∈ F do
4       f' ← Mutate(f)
5       feedback ← c.run(f')
6       if feedback > 0 then
7         F ← F ∪ {f'}
8   T ← F
9   return T
```

B. Feedback-guided strategy

Circuit Structure Mutation is an approach based on random mutations. However, the random mutation strategy may be not very efficient for some purposes, such as increasing test coverage and find performance regressions. It is because the mutation process has no guidance to achieve the purposes. Therefore, we propose a feedback-guided strategy to guide the mutation process towards the purpose. The main idea of the feedback-guided strategy is to monitor the feedback of running the mutant on the hardware model checkers and put the mutant into the seed files if the feedback is positive. For example, if the purpose is to increase the test coverage, the mutants that cover new code are added to the seed files to be mutated in later iterations. If the purpose is to find performance regressions, the mutants that trigger minor performance regressions are added to the seeds in order to generate major performance regressions based on them.

Algorithm 2 describes the feedback-guided strategy in `Hammer`. In this algorithm, inputs are the seed files F and the checker c used for providing coverage information. First,

Algorithm 3: Differential testing process

```
   Input: Seed files F, Hardware model checkers C
   Output: issues
1  Procedure DifferentialTest(F, C):
2      issues ← ∅
3      testsuite ← FeedbackMutation(C, F)
4      forall test ∈ testsuite do
5          result_ref ← null
6          forall c ∈ C do
7              result ← c.run(test)
8              if result_ref = null or crash then
9                  result_ref ← result
10             if result = crash then
11                 issues ← issues ∪ {(test, c)}
12             else if result ≠ result_ref then
13                 issues ← issues ∪ {(test, c)}
14     return issues
```

there is a while-loop on the seed set F that repeats forever unless a timeout or user interruption occurs (lines 2 and 3). In each iteration, for each file f in F, f is mutated to f' by Algorithm 1 (line 4). A *feedback* is obtained by running checker c on the mutant f' (line 5). If *feedback* is positive, the mutant f' is appended to seeds F (lines 6 and 7). At last, this process returns the updated seeds as the test suite (line 9).

In practice, the design of the feedback function will affect the effectiveness of *Circuit Structure Mutation*. In `Hammer`, the feedback is positive when the mutant covers new paths if the purpose is increasing the test coverage. If the purpose is to find performance regressions, the feedback is positive when the performance difference between two hardware model checkers is larger than ten seconds. Users could design their own feedback function according to their purposes.

C. Differential Testing Framework

To obtains the test oracles of the test inputs generated by the feedback-guided *Circuit Structure Mutation*, we adopt a differential testing framework. Differential testing is a widely used technique for detecting bugs in software. The key insight is to compare the results of different implementations for the same task, if the results are not the same, there should be an issue in one of the implementations. In `Hammer`, we use differential testing to find both unsoundness issues and performance regressions. For unsoundness issues, if two hardware model checker implementations report different results or one hardware model checker implementations crash while the other does not, we think this test input triggered an unsoundness issue in one of the checker implementations. For performance regression, if one model checker implementation can give the result in a short time while the other one solves the same problem in a significantly longer time, we think this test input triggered a performance regression in the latter implementation. This differential strategy could also be used to find performance optimization chances by comparing the execution times between two hardware model checkers.

Algorithm 3 presents our differential testing process. The inputs are the seed files F and the hardware model checker set C used for differential testing. The checker set C can consist of more than two items. The checker set C provides feedback for `FeedbackMutation` to generate test suites $testsuite$ (line 4). Then, the process runs each c in C on the test inputs $testsuite$ (line 6 and 7). If a hardware model checker crashes (line 10) or the hardware model checkers give different results (line 12 with lines 8 and 9), the test inputs and the corresponding checkers will be recorded in $issues$. Here, the definition of the equivalence between $result$ and $result_{ref}$ can be defined according to the purpose. For example, for unsoundness issue, we could define if one of the results is unknown or timeout, or two results are both safe or unsafe, these two results are equal. For performance regression, we could define if one hardware model checker implementation can solve in one second, while the other one takes more than 10 seconds, these two results are not equal. Finally, The issue triggering set $issues$ will then be manually investigated and de-duplicated. For unsoundness issues, we could verify the counterexample reported by the hardware model checker that gives unsafe via simulation, if the counterexample is bogus, the issue should be in the checker that reports unsafe, otherwise, the issue should in the other one.

D. Implementation of `Hammer`

Fig. 2 presents the overall framework of `Hammer`. Given a group of seed files, a file is randomly selected from the seed files and put into the *Circuit Structure Mutation* process. Then, according to Algorithm 1, *Circuit Structure Mutation* generates mutant using the seed file. The mutant is then fed into the differential testing framework. This framework compares the execution results of the hardware model checkers. If this mutant triggers interesting cases, such as performance optimization chance and reliability issue, or it can increase the code coverage, the framework will save this mutant as the test input into the test suite, which was described in Algorithm 3. On the other hand, the differential testing framework also provides the information for feedback-guided strategy. If the mutant triggers a significant performance difference between two hardware model checkers in the differential testing framework or the mutant can increase the test coverage in one hardware model checker, the differential testing framework will provide positive feedback to the feedback-guided strategy. If the feedback is positive, the mutant will be added as the new seed into the seed files and be selected in the future iteration. The whole process will iterate until the user terminates `Hammer` or expiring the timeout configuration.

The mutation component of the `Hammer` was implemented in C programming language and it can be executed independently for testing hardware model checkers without feedback-guided strategy. The coverage-guided part of `Hammer` inherits from `AFL` which is implemented in C++ programming language. We removed all native mutation strategies in `AFL`

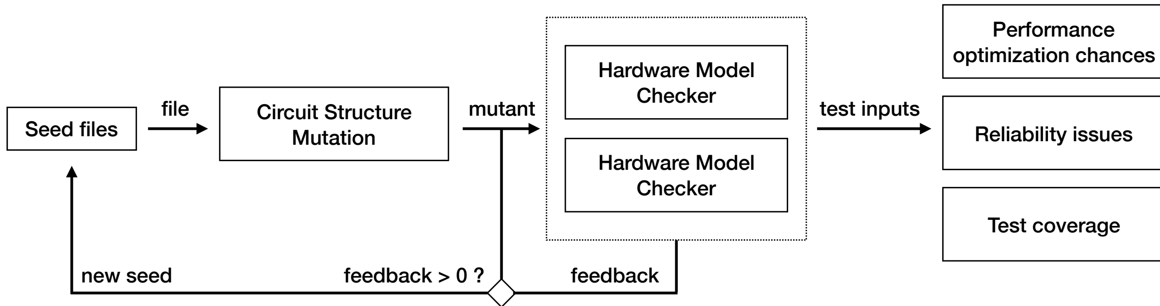


Fig. 2: Framework of `Hammer`.

```
1 aag 1 0 1 0 0 1
2 2 0 2
3 2
```

Fig. 3: AIG test input for triggering ABC unsoundness issue.

```
1  aag 41 0 22 0 19 1 3
2  2 0
3  4 0
4  6 0
5  8 0
6  10 0
7  12 0
8  14 0
9  16 0
10 18 0
11 20 0
12 22 0
13 24 0
14 26 0
15 28 0
16 30 0
17 32 0
18 34 0
19 36 36
20 38 38
21 40 40
22 42 42
23 44 44
24 1
25 52
26 66
27 82
28 46 38 36
29 48 46 40
30 50 48 42
31 52 50 44
32 54 5 3
33 56 54 7
34 58 56 9
35 60 58 11
36 62 60 13
37 64 62 15
38 66 64 17
39 68 21 19
40 70 68 23
41 72 70 25
42 74 72 27
43 76 74 29
44 78 76 31
45 80 78 33
46 82 80 35
```

Fig. 4: AIG test input for triggering ABC segmentation fault.

and made the mutation stage call our mutation core, which leverages the `AFL` coverage-guided infrastructure to benefit our mutation process. The performance-guided infrastructure is implemented in shell script. The generated test inputs that trigger interesting cases or increase test coverage are stored in a folder. The differential testing framework in Go language reads the folder and validates the hardware model checkers.

V. Evaluation

A. Evaluation Setup

To evaluate the effectiveness of `Hammer`, we designed three experiments for evaluating its bug-finding ability, test coverage performance, and performance-regression-finding ability respectively. The experiments ran on a computer with Intel i7-8700 CPU, 8GB RAM, and Ubuntu 18.04 operating system.

Bug-finding ability. In this experiment, we investigate whether the vanilla `Hammer` (i.e., without feedback-guided strategy) can find bugs in hardware model checkers as we expected. We select three state-of-the-art hardware model checkers ABC [8], nuXmv [9], and AVY [10] as the targets being tested. The seeds come from the benchmarks on AIGER website [29] and HWMCC 2017 [7]. The seeds include 3,610 files that span from AIGER old format to AIGER-1.9.4 format. Before the evaluation, we ran all three hardware model checkers through the seeds to ensure none of the seeds trigger issues in these three targets. The timeout for each checking is 30 seconds.

Test coverage performance. In this experiment, we investigate whether the mutants generated by `Hammer` with coverage-guided feedback can increase line, function, branch, and path coverage. Here, we define the feedback function as if the mutant can cover the new path that was not covered by the previous test inputs, the feedback is positive. We evaluate the code and path coverage on two open-source model checkers ABC and AVY. The evaluation period is 6 hours for each checker. We use pdr model checking engine for ABC. The seeds used for this evaluation are sampled from the benchmarks above. To fully squeeze the seeds in 6 hours, we build up the seeds with 50 sampled benchmarks that can be solved in 30 seconds. The code coverage was obtained by GCov [30] tool and the numbers of explored paths are recorded by the `AFL` framework. We compare the feedback-guided `Hammer` with `aigfuzz`, `AFL` and the `Hammer` without feedback-guided strategy (`Hammer-noCov`).

Performance-regression-finding ability. In this experiment, we investigate whether the mutants generated by `Hammer` with performance-guided feedback can find performance regressions that are not uncovered before. Specifically, the goal of the `Hammer` in this experiment is to find the test inputs that trigger significant performance differences between ABC and AVY. Here, the significant performance difference means the same test input triggers more than 10 seconds execution

TABLE I: Issues found by `Hammer`.

ID	Tool	Type	Triggering command	Location (if known)
1	ABC	unsoundness	abc -c "pdr" <aig>	-
2	ABC	assertion failure	abc -c "pdr" <aig>	satClause.h
3	ABC	assertion failure	abc -c "int" <aig>	intCore.c
4	ABC	assertion failure	abc -c "tempor" <aig>	aigMem.c
5	ABC	segmentation fault	abc -c "fold;bmc2" <aig>	bmcBmc2.c
6	ABC	segmentation fault	abc -c "fold;bmc3" <aig>	-
7	AVY	unsoundness	avy <aig>	-
8	AVY	assertion failure	avy <aig>	vecWec.h
9	AVY	segmentation fault	avy <aig>	AigUtils.h
10	nuXmv	segmentation fault	check_property	-

time difference between two hardware model checkers. The timeout for each checking is 15 seconds. Note that if one hardware model checker terminated in 5 seconds and the other one ran into timeout, we think this test input also triggers a significant performance difference. We also use pdr model checking engine for ABC. The feedback function is defined as if the mutant triggers one more second execution time difference between two hardware model checkers, the feedback is positive. In this experiment, we collected the 243 benchmarks that can be solved by both ABC and AVY in one second as the seed files. If performance-guided `Hammer` could find significant performance difference using these seed files, it will show the effectiveness of feedback-guided *Circuit Structure Mutation* on finding performance regressions. The evaluation period is 12 hours. We compare the feedback-guided `Hammer` with `aigfuzz` and the `Hammer` without feedback-guided strategy (`Hammer-noPerf`).

B. Evaluation Results

Bug-finding ability. In this experiment, `Hammer` found 10 unique issues in ABC, nuXmv and AVY. Table I presents the details of the issues. The first column shows the names of the hardware model checkers that contain the issues. The second column specifies the types of the issues: "unsoundness" refers to this issue that makes the checker produce an incorrect checking result, "assertion failure" or "segmentation fault" refers to the crash for different reasons. The third column gives the commands for triggering the issues and the fourth column gives the location of issues. Each issue can be found many times in the evaluation, we have carefully de-duplicated the issue triggers to avoid duplicate, while the segmentation fault in nuXmv is difficult to de-duplicate due to the lack of debugging information. We next present some reduced examples of the issues found in the experiment.

The AVY unsoundness issue was presented in Fig. 1c, we have reported this issue to the developers and it has not been fixed yet. If we remove the invariant constraint in Fig. 1c, the test input will trigger the assertion failure in AVY. Next, we show two bug samples in ABC. Fig. 3 presents a test input that triggers an unsoundness issue in ABC. This small test input was reduced from a large AIG file that triggers the same issue. It should be unsafe, while ABC reports safe. It is due to some ABC commands assume that the latch is zero-initialized and this corner case uncovers this unsound assumption by building a latch that is initialized by itself. Fig. 4 presents a test input that triggers ABC segmentation fault with the command "abc -c "fold;bmc3" <aig>" which calls the bounded model checking engine of ABC. The segmentation fault is due to the significant memory usage while checking this case, however, using other bounded model checking engines, such as "bmc2", will not encounter the segmentation fault. The rest issue-triggering files are too large, so we do not incorporate them into this paper.

The evaluation results show that `Hammer` is effective in finding issues in hardware model checkers. The issues are diverse and mostly related to the hardware model checker usability. The issues in hardware model checkers are difficult to be found, since the code base of the core components is small, which is unlikely to contain many issues. Besides, in the evaluation, we observed that some mutants of `Hammer` make the checkers take significant memory usage, while the seeds do not, which shows the potential usage of `Hammer` for finding performance issues and evidences that the mutants give more stress on the checkers than the seeds.

Test coverage performance. The results of this experiment are presented in Table II. The first four lines in Table II present the numbers of lines, functions, branches, and paths covered by the seeds, `aigfuzz`, `AFL`, `Hammer-noCov` and `Hammer` respectively. We first investigate the code coverage matrices. Table II shows that the lines, functions, and branches explored by `AFL`, `Hammer-noCov`, and `Hammer` are consistently higher than the seeds, which means the mutation-based strategies can generate more diverse test inputs than the seeds. The highest numbers for each matrix are highlighted in grey. Although the differences are small, it still shows that `Hammer` performs best on most matrices. One possible explanation of the minor code coverage increments is the checking algorithms in hardware model checkers are highly recursive. Thus, the same lines, branches, and functions are called repeatedly during the checking process. However, different test inputs could trigger different paths. We hence investigate the path coverage. Table II provides the number of paths explored by the seeds, `AFL` and `Hammer` respectively. The 50 seeds initially explore 50 paths, and the mutation-based strategies explore much more paths than the seeds. According to the results, the numbers of paths explored by `Hammer` are significantly higher (about 25% increment) than others, which shows the strength of `Hammer`. We also compare our mutation-based fuzzer `Hammer` with the

TABLE II: Numbers of lines, functions and branches covered by the tools in one hour.

	ABC (pdr)				AVY			
	lines	func.	bran.	paths	lines	func.	bran.	paths
seeds	6,082	495	3,042	50	13,946	3,256	10,766	50
`aigfuzz`	6,019	498	2,991	1,213	13,835	3,252	10,660	2,185
`AFL`	6,208	503	3,141	1,295	14,004	3,262	10,847	2,812
`Hammer-noCov`	6,210	504	3,134	1,301	14,048	3,266	10,888	2,926
`Hammer`	6,210	504	3,134	1,604	14,049	3,266	10,887	3,520

TABLE III: Numbers of significant performance differences found in ABC (pdr) and AVY by `Hammer` in 12 hours.

Tool	#total	#ABC	#AVY
`aigfuzz`	12	0	12
`Hammer-noPerf`	291	154	137
`Hammer`	324	199	125

generation-based fuzzer `aigfuzz`, which is presented in the second line of Table II. We ran `aigfuzz` with the default configuration for 6 hours and obtained the code and path coverage. The results of `aigfuzz` show that the generation-based testing tool covers fewer lines, functions, and branches than the mutation-based testing tool and their seeds in 6 hours, which evidences the general advantage of mutation-based testing tool on testing hardware model checkers: high-quality seeds provide a nice starting point, thus the mutants can be more effective for exploring the code than the randomly generated test inputs.

In summary, `Hammer` significantly outperforms the seeds and `AFL` regarding the number of explored paths, which shows the effectiveness of *Circuit Structure Mutation* and partially explains why `Hammer` can find the issues that cannot be triggered by the seeds. The increment from `Hammer-noCov` to `Hammer` shows the contribution of coverage guidance. Even though without coverage information, `Hammer` achieved close code exploration numbers comparing to coverage-guided `AFL`, which shows the strength of *Circuit Structure Mutation*.

Performance-regression-finding ability. Table III presents the results for this experiment. The second column refers to the total number of significant performance differences found by the corresponding tool in 12 hours. We can see that `aigfuzz` can only find 12 significant performance differences while both `Hammer-noPerf` and `Hammer` found more than one hundred significant performance differences in 12 hours, which shows the effectiveness of *Circuit Structure Mutation* on finding the performance regression in hardware model checkers. With feedback-guided strategy, `Hammer` can find 30 more significant performance differences, which shows the effectiveness of the feedback-guided strategy. Then, we would like to know whether `Hammer` can find performance optimization chances (*i.e.*, one checker slower than other) in both checkers. First, in Table III, we can see that `aigfuzz` can only find the optimization chances in AVY, while `Hammer-noPerf` and `Hammer` can find the optimization chances in both ABC and AVY. As ABC had better overall performance than AVY in the previous competition [7], the optimization chances in ABC will be more interesting. We can note that `Hammer` can find more optimization chances in ABC than `Hammer-noPerf` and no much fewer optimization chances in AVY.

In summary, `Hammer` shows its effectiveness on finding performance regressions by finding many significant performance differences between ABC and AVY using the seeds that take only one second to solve. The increment from `Hammer-noPerf` to `Hammer` shows the contribution of performance guidance. Even though without performance guidance, `Hammer-noPerf` can significantly outperform the only existing tool `aigfuzz` on finding the performance differences. This experiment shows that the developers of hardware model checker can use `Hammer` to find the performance regression in their developing process.

VI. Conclusion

In this paper, we have proposed *Circuit Structure Mutation*, a simple but effective approach for generating test inputs for hardware model checkers and implemented a feedback-guided mutation-based testing tool `Hammer`. This tool can be used to find reliability bugs, increase test coverage and find performance regressions in hardware model checkers. In our evaluation, `Hammer` shows its effectiveness. First, `Hammer` found 10 issues in the state-of-the-art hardware model checkers, which shows its ability on finding bugs. On the aspect of test coverage, `Hammer` achieved more test coverage than the previous tools `AFL` and `aigfuzz`. Last, it can efficiently uncover the significant performance difference between two hardware model checkers, which shows its ability on finding performance regressions and optimization chances.

Besides, `Hammer` has wide potential usages. For example, it can be used by the hardware model checking community to validate the correctness and reliability of the implementation, detect potential performance issues, and generate benchmarks for HWMCC. Meanwhile, the key idea of *Circuit Structure Mutation* could be also adapted to test other hardware design and verification tools. It will be an interesting future work to extend *Circuit Structure Mutation* beyonds AIG input format.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. We appreciate Alan Mishchenko and Arie Gurfinkel for answering the questions related to ABC and AVY. Jianwen Li is supported in part by Shanghai Pujiang Talent Plan No. 20PJ1403500 and NSFC No. 62002118. Ting Su is supported by NSFC No. 62072178 and the project of STC of Shanghai No. 19511103602. Geguang Pu is supported by National Key R&D Program of China No. 2020AAA0107800.

References

[1] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[2] E. M. Clarke, E. A. Emerson, and J. Sifakis, "Model checking: algorithmic verification and debugging," *Communications of the ACM*, vol. 52, no. 11, pp. 74–84, 2009.

[3] E. Clarke, A. Gupta, H. Jain, and H. Veith, "Model checking: Back and forth between hardware and software," in *Working Conference on Verified Software: Theories, Tools, and Experiments*, 2005, pp. 251–255.

[4] A. Goel and K. Sakallah, "Empirical evaluation of ic3-based model checking techniques on verilog rtl designs," in *DATE*, 2019, pp. 618–621.

[5] J. Jörg Bormann, J. Lohse, M. Payer, and G. Vezin, "Model checking in industrial hardware design," in *DAC*, 1995, pp. 298–303.

[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *CAV*, 2002, pp. 359–364.

[7] A. Biere, T. van Dijk, and K. Heljanko, "Hardware model checking competition 2017," in *FMCAD*, 2017, pp. 9–9.

[8] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *CAV*, 2010, pp. 24–40.

[9] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *CAV*, 2014, pp. 334–342.

[10] Y. Vizel and A. Gurfinkel, "Interpolating property directed reachability," in *CAV*, 2014, pp. 260–276.

[11] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *TSE*, 2019.

[12] American fuzzy lop (2.52b). [Online]. Available: https://lcamtuf.coredump.cx/afl/

[13] Mozilla firefox. [Online]. Available: https://www.mozilla.org/firefox/

[14] Mysql. [Online]. Available: https://www.mysql.com/

[15] Openssl. [Online]. Available: https://www.openssl.org/

[16] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[17] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2017.

[18] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *TSE*, 2019.

[19] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *ICSE*, 2019, pp. 724–735.

[20] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars," in *NDSS*, 2019.

[21] C. Zhang, T. Su, Y. Yan, F. Zhang, G. Pu, and Z. Su, "Finding and understanding bugs in software model checkers," in *ESEC/FSE*, 2019, pp. 763–773.

[22] C. Klinger, M. Christakis, and V. Wüstholz, "Differentially testing soundness and precision of program analyzers," in *ISSTA*, 2019, pp. 239–250.

[23] R. Brummayer, F. Lonsing, and A. Biere, "Automated testing and debugging of sat and qbf solvers," in *SAT*, 2010, pp. 44–57.

[24] D. Winterer, C. Zhang, and Z. Su, "On the unusual effectiveness of type-aware operator mutations for testing smt solvers," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–25, 2020.

[25] ——, "Validating smt solvers via semantic fusion." in *PLDI*, 2020, pp. 718–730.

[26] M. N. Mansur, M. Christakis, V. Wüstholz, and F. Zhang, "Detecting critical bugs in smt solvers using blackbox mutational fuzzing," in *ESEC/FSE*, 2020.

[27] D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, and V. Ganesh, "Stringfuzz: A fuzzer for string solvers," in *CAV*. Springer, 2018, pp. 45–51.

[28] J. Scott, F. Mora, and V. Ganesh, "Banditfuzz: A reinforcement-learning based performance fuzzer for smt solvers," in *Software Verification*. Springer, 2020, pp. 68–86.

[29] Aiger. [Online]. Available: http://fmv.jku.at/aiger/

[30] Gcov. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html