Specification-Driven Conformance Checking for Virtual/Silicon Devices using Mutation Testing

Haifeng Gu, Jianning Zhang, Mingsong Chen, Senior Member, IEEE, Tongquan Wei, Member, IEEE, Li Lei, Member, IEEE, and Fei Xie, Member, IEEE

Abstract—Modern software systems, either system or application software, are increasingly being developed on top of virtualized software platforms. They may simply intend to execute on virtual machines or they may be expected to port to physical machines eventually. In either case, the devices, virtual or silicon, in the target virtual or physical machines are expected to conform to the specifications based on which the software systems have been developed. Non-conformance of these devices to the specifications can cause catastrophic failures of the software systems. In this paper, we propose a mutation-based framework for effective and efficient conformance checking between virtual/silicon device implementations and their specifications. Based on our defined mutation operators, device specifications can be automatically instrumented with weak mutant-killing constraints to model potential erroneous device behaviors. To kill all feasible mutants, our approach adopts a cooperative symbolic execution mechanism that can efficiently automate the test case generation and conformance checking for virtual/silicon devices. By symbolically executing the instrumented specifications with virtual/silicon device traces obtained from the cooperative execution, our method can accurately measure whether the designs have been sufficiently validated and report the inconsistencies between device specifications and implementations. Comprehensive experiments on two industrial network adapters and their virtual devices demonstrate the effectiveness of our proposed approach in conformance checking for both virtual and silicon devices.

Index Terms—Conformance Checking, Mutation Testing, Virtual Prototype, Silicon Device, Specification.

1 INTRODUCTION

VIRTUALIZATION has not only been a revolutionary technique in deploying software systems, but also begun to play a crucial role in speeding up software development [1]. Software systems are increasingly being developed on top of virtualized system platforms, that is, emulating real silicon devices (e.g., buses, accelerators, network adapters) within a virtual machine (VM). These software systems may intend to execute just on these VMs or they may eventually be ported to the physical machines emulated by these virtual platforms. To ensure these software systems correctly execute in the targeted deployment platforms, virtual or silicon, it must be established that the virtual or silicon devices with which the software systems interact indeed conform to their specifications; otherwise, the software systems can fail catastrophically as they execute on these devices.

To ensure these devices conform to their specifications, two key challenges need to be addressed. The first challenge is the lack of automated testing approaches that can sufficiently validate whether all the desired functionalities modeled in specifications are correctly implemented in virtual/silicon devices. Although traditional testing approaches can potentially achieve expected functional coverage for virtual devices, such coverage information cannot fully reflect the real interactions between hardware and software components. The situation becomes even worse when dealing with the black-box silicon devices. The second challenge is the lack of effective conformance checking tools to identify design inconsistencies between different abstraction layers. If virtual and silicon devices do not always conform to each other, drivers developed on the virtual system platforms often cannot readily work on silicon devices. The silicon device errors or driver bugs eclipsed by incorrect virtual devices may cause serious problems, e.g., system crashes and blue screens [2], [3].

As the de-facto device interface specification language, SystemRDL [4] has been widely adopted in virtual/silicon device design to model their register structures. However, the current version of SystemRDL does not support the behavioral modeling of register accesses. As an alternative, an executable version of SystemRDL, namely xSystemRDL, has been proposed in [5] by extending both the syntax and semantics of SystemRDL. Based on the C programming language, xSystemRDL specifications allow designers to specify high-level register access behaviors over the defined registers. Since xSystemRDL can accurately specify hardware/software interactions in terms of register accesses, it can be used as a golden reference model for the purpose of conformance checking both virtual and silicon devices. In [5], a framework has been proposed to automatically translate xSystemRDL into a Formal Device Model (FDM) that has complete formal semantics and is amenable to symbolic analysis.

In this paper, we propose a novel mutation-driven framework that can automatically generate effective test cases for the conformance checking between high-level specifications (i.e., FDMs) and low-level implementations (i.e., virtual and silicon devices). Our approach is different from traditional mutation testing approaches that kill only one mutant for each mutated program. Instead, inspired by [6], we instrument all the generated Mutant-
Section 4 presents experimental results on two industrial network testing efforts.

 achieves better test coverage than state-of-the-art methods with less responded silicon devices widely used in industry, but also can the open source machine virtual platform QEMU [7] and cor-

 effectively uncover bugs from both virtual devices excerpted from experimental results show that our approach can not only ef-

 ciently generate mutation-based test cases. By reducing the killing of mutant-instrumented code, mutation testing [19], [20], [21]. In [44], Ammann et al. proposed a way of computing the size of the minimal mutant set for mutant set minimization with respect to a specific test set. To reduce the test data generation time, various automated mutation testing approaches have been proposed. For example, Papadakis and Malevris [22] proposed a novel approach that conjoins program transformation and dynamic symbolic execution techniques to automatically generate mutation-based test cases. By reducing the killing mutants’ problem into a branch-coverage one, they combined symbolic execution, concolic execution, and evolutionary testing methods to produce test cases according to the weak mutation testing criterion in [23]. In [6], Zhang et al. proposed a mutation testing approach called PexMutator. By transforming a program into an instrumented meta-program that contains mutant-killing constraints, PexMutator adapts dynamic symbolic execution to automate the test case generation to kill instrumented mutants. Although these approaches can generate high-quality test inputs, few of them consider the mutation testing for specific design features of virtual/silicon devices.

 Along with the prosperity of virtualization techniques and tools (e.g., QEMU [7], VMWare, VirtualBox and Xen), the use of virtual devices in place of physical hardware is increasing in computer system design and testing, since they have better observability and controllability [24], [25], [26]. For example, SimTester [27], SIMEXPLORER [28] and SDRacer [29] are promising frameworks that can facilitate the testing and debugging of interrupt-driven embedded software. Instead of running software directly on real hardware devices, these approaches employ virtual platforms that can precisely control execution events and observe runtime context at critical code locations. However, all these methods focus on embedded software rather than the underlying host devices. To check the correctness of manually developed virtual devices, Yu et al. [30] proposed a novel framework for testing virtual devices within a full system simulator. By using physical devices to eliminate the need for manual test oracles, their approach can detect more faults than random testing. However, their approach does not consider conformance checking and cannot be directly applied on silicon devices due to limited observability.

 To enable the conformance checking between different abstraction layers of computer systems, various specification-driven methods have been investigated [8], [9]. For example, Bombieri et al. [10] presented an event-based approach that supports the equivalence checking between Transaction Level Modeling (TLM) and Register Transfer Level (RTL) designs. By checking the activation order of corresponding Property Specification Language-based assertions in both TLM and RTL designs, Chen et al. [11] proposed an approach that can enhance the observability of conformance checking. Based on timed automata, Herber et al. [12] introduced a conformance testing method to check the consistency between abstract models and SystemC implementations. However, none of the above approaches enable the conformance checking for virtual and silicon devices.

 As a promising program analysis technique, symbolic execution has been widely used in the testing of software and hardware components. For example, SAGE [13] and S2E [14] are two popular testing tools based on symbolic execution for software systems that intensively interact with external environments. By combining the dynamic symbolic execution and constraint solving techniques, the tools KLEE [15], CUTE [16] and CRETE [17] can automatically generate high-quality test cases for both hardware and software designs. In [18], Guo et al. proposed a promising testing approach that can convert multi-task PLC programs into C code for the test case generation based on KLEE. However, so far most symbolic execution techniques focus on testing rather than conformance checking.

 By seeding artificial defects into programs, mutation testing can be used to assess and improve the quality of a given test suite based on the number of killed (identified) mutants [19], [20], [21]. In [44], Ammann et al. proposed a way of computing the size of the minimal mutant set for mutant set minimization with respect to a specific test set. To reduce the test data generation time, various automated mutation testing approaches have been proposed. For example, Papadakis and Malevris [22] proposed a novel approach that conjoins program transformation and dynamic symbolic execution techniques to automatically generate mutation-based test cases. By reducing the killing mutants’ problem into a branch-coverage one, they combined symbolic execution, concolic execution, and evolutionary testing methods to produce test cases according to the weak mutation testing criterion in [23]. In [6], Zhang et al. proposed a mutation testing approach called PexMutator. By transforming a program into an instrumented meta-program that contains mutant-killing constraints, PexMutator adapts dynamic symbolic execution to automate the test case generation to kill instrumented mutants. Although these approaches can generate high-quality test inputs, few of them consider the mutation testing for specific design features of virtual/silicon devices.

## 2 Related Works

To enable the conformance checking between different abstraction layers of computer systems, various specification-driven methods have been investigated [8], [9]. For example, Bombieri et al. [10] presented an event-based approach that supports the equivalence checking between Transaction Level Modeling (TLM) and Register Transfer Level (RTL) designs. By checking the activation order of corresponding Property Specification Language-based assertions in both TLM and RTL designs, Chen et al. [11] proposed an approach that can enhance the observability of conformance checking. Based on timed automata, Herber et al. [12] introduced a conformance testing method to check the consistency between abstract models and SystemC implementations. However, none of the above approaches enable the conformance checking for virtual and silicon devices.

As a promising program analysis technique, symbolic execution has been widely used in the testing of software and hardware components. For example, SAGE [13] and S2E [14] are two popular testing tools based on symbolic execution for software systems that intensively interact with external environments. By combining the dynamic symbolic execution and constraint solving techniques, the tools KLEE [15], CUTE [16] and CRETE [17] can automatically generate high-quality test cases for both hardware and software designs. In [18], Guo et al. proposed a promising testing approach that can convert multi-task PLC programs into C code for the test case generation based on KLEE. However, so far most symbolic execution techniques focus on testing rather than conformance checking.

By seeding artificial defects into programs, mutation testing can be used to assess and improve the quality of a given test suite based on the number of killed (identified) mutants [19], [20], [21]. In [44], Ammann et al. proposed a way of computing the size of the minimal mutant set for mutant set minimization with respect to a specific test set. To reduce the test data generation time, various automated mutation testing approaches have been proposed. For example, Papadakis and Malevris [22] proposed a novel approach that conjoins program transformation and dynamic symbolic execution techniques to automatically generate mutation-based test cases. By reducing the killing mutants’ problem into a branch-coverage one, they combined symbolic execution, concolic execution, and evolutionary testing methods to produce test cases according to the weak mutation testing criterion in [23]. In [6], Zhang et al. proposed a mutation testing approach called PexMutator. By transforming a program into an instrumented meta-program that contains mutant-killing constraints, PexMutator adapts dynamic symbolic execution to automate the test case generation to kill instrumented mutants. Although these approaches can generate high-quality test inputs, few of them consider the mutation testing for specific design features of virtual/silicon devices.

## 3 Our Conformance Checking Approach

As a promising approach to evaluate and improve the quality of test cases, mutation testing [19], [34], [42] tries to distinguish the
original program under test from its various faulty variants (a.k.a., mutants). A test case can kill a mutant if its executions on the mutant and its original program can produce different results (i.e., final states). The quality of test cases can be judged by the number of mutants killed by the test cases. This idea can be naturally borrowed in conformance checking between specifications and implementations of virtual/silicon devices. During conformance checking, we can consider the implementation inconsistencies as special implementation-level mutations caused by improper design refinement, which can be reflected by corresponding mutants in the specification level. The test cases aiming to kill such mutants can be used to check whether the functional scenario of the specification is correctly implemented in virtual/silicon devices.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(a,b);</td>
<td>read(a,b);</td>
</tr>
<tr>
<td>MutantCheck((a+b)!=(a-b));</td>
<td>b=a+b;</td>
</tr>
<tr>
<td>b=a+b;</td>
<td>b=a-b;</td>
</tr>
<tr>
<td>return b;</td>
<td>return b;</td>
</tr>
</tbody>
</table>

Fig. 1. A motivating example of mutation-based conformance checking

Figure 1 presents an example to explain this idea. The left part of the figure gives the specification of a design, which tries to return the addition of two input registers (i.e., a and b). In this example, we assume that the statement “b=a+b;” is implemented with “b=a-b;” by mistake as shown in the right part. When the inputs of both a and b equal 0, it is hard to identify the inconsistency. To avoid this, we add an MKC (the conditional statement wrapped in MutantCheck()) to guide the test case generation. If an FDM test case (e.g., a = 1, b = 1) can trigger the MutantCheck() function, it should satisfy the wrapped MKC (i.e., (a+b)!=(a-b)). If the implementation is incorrectly implemented as such, the generated FDM test case can be used to differentiate between the specification and implementation for the addition operation. It is worth noting that the instrumented MKCs do not change any behaviors of the original specification since they are conditional checks rather than assignments.

Fig. 2. Workflow of our conformance checking approach

Figure 2 shows the workflow of our approach, which includes three major components: mutant generator, test case generator and conformance checker. In our approach, we use FDM that is automatically generated from xSystemRDL as the golden reference specification for device implementation. Based on the given mutation operators (see Section 3.1), our mutant generator can automatically instrument a given FDM with MKCs. Unlike traditional mutation testing that generates one program for each mutant, our approach encodes each mutant using an MKC and instruments all the MKCs within the given FDM. Based on symbolic execution, our test case generator (see Section 3.2) tries to kill all the specified mutants by covering the true branches of their conditional statements. Note that to directly operate devices our approach manages interface registers by proper synchronization without resorting to device drivers. By symbolically running the collected traces generated from device testing on the instrumented FDM, our conformance checker (see Section 3.3) can report the inconsistency based on our defined conformance rules. The following subsections will introduce our approach in detail.

### 3.1 Mutant-Killing-Constraint Generation and Instrumentation

Mutation operators play an important role in mutation testing, since their rules can be used to enable the automated mutant generation. To adequately conduct conformance checking, it is required to generate a rich set of mutants to explore all potential inconsistencies. However, in most cases increasing the types of mutation operators does not lead to additional benefits but wasted time and resources [19], [34]. To accommodate our conformance checking purpose, our approach focuses on six selected mutation operators related to interface registers as shown in Table 1. It is important to note that these six mutation operators are by no means the “golden” ones rather they are considered as a set of practically useful operators [34] for the mutation testing of traditional software. In practice, our approach welcomes other types of mutation operators, which can be easily integrated into our framework to strengthen the detection capability of new inconsistencies.

As shown in the first column of Table 1, the top four mutation operators come from the sub-set of mutation operators (i.e., ABS, AOR, LCR, ROR and UOI) proposed by Offutt et al. [34]. Our approach does not consider the operator Absolute Value Insertion (i.e., ABS), since negative values are not involved in dealing with interface registers. Moreover, since the assignments of reserved registers and register read operations are important for interface registers, we create two new mutation operators to model the possible errors. Column 2 presents the rules for the mutations. Each rule consists of two parts delimited by “→”, where the left part denotes the correct design in a specification and the right part indicates the potential errors. By default, \( RO = \{,<,=,>,>=,==,!=\} \), \( LC = \{\&,\|\} \), \( AO = \{+,-,*,/,\%\} \), and \( UO = \{\sim\} \). For example, by using the Arithmetic Operator Replication (AOR) operator we can replace the arithmetic operator \( \alpha \) with a new one \( \beta \), e.g., replacing \( a + b \) with \( a - b \). For the last two rows, RRVR tries to assign each reserved register \( reg \) with a special value defined within \( D = \{0xfffff, 0x00000000, 0x84218421\} \), and RRAI tries to assign each read register with a specific value from \( VAL = \{0xfffff\} \). The values in \( D \) and \( VAL \) indicate the most common corner cases that can potentially cause inconsistencies between FDMs and virtual/silicon devices. Note that both \( D \) and \( VAL \) can be extended or redefined for different testing purposes. Since FDMs are automatically translated from xSystemRDL specifications, the reserved register information (i.e., \( RSV \) in RRVR) can be collected during the translation process for the following MKC generation. The register-read-access operation can be figured out during the FDM parsing, where the values of registers are returned within return statements.

Instead of using strong mutant killing for test generation which is intractable in practice [35], our approach adopts the weak mutation testing [6], [36] to improve the probability of mutant killing. In our approach, the test case generation is guided by killing
the specification mutants generated by our mutation operators, which model a set of potential inconsistencies implemented in virtual/silicon devices. Based on the same interface registers, a generated test case involving a sequence of device requests can be used for conformance checking. Let $S$ and $M_{spec}$ denote an FDM-based specification and a mutant of $S$ on statement $st$ (the other parts of $S$ and $M_{spec}$ are the same), respectively. A test input $t$ can weakly kill $M_{spec}$ if the following two criteria can be satisfied: i) for reachability, $t$ must trigger $st$ on both $S$ and $M_{spec}$; and ii) for necessity, the internal states of $M$ and $M_{spec}$ must be different immediately after executing $st$. Our approach does not require the sufficiency, which implies that the final states of $S$ and $M_{spec}$ after executing $t$ are different.

Similar to [6], our approach adopts the MKCs to efficiently enable weak mutant killing, thus facilitating the symbolic execution-based test case generation. Instead of generating one specification for each mutant, our approach encodes each mutant using an MKC and instruments them within the target FDM. Note that all these MKCs only contain conditional statements, which will not alter the semantics of original specifications. In this way, if an MKC is satisfied by a test input, the test case can be used to kill the corresponding mutant.

The third and fourth columns of Table 1 present the rules and examples for the MKC generation, respectively. Similar to the notation “$\rightarrow$” used in the second column, the rules for MKC generation in column 3 also have two parts. The left part denotes the specification constructs, while the right part denotes the generated conditional statements wrapped in $MutantCheck()$ function. Note that one mutation operator may lead to the generation of multiple MKCs. For example, since $AO = \{+,−,+,/,\%\}$ has five elements, when dealing with the statement $c = a \pm b$, four MKCs will be generated as follows: $MutantCheck((a + b)! = (a - b))$, $MutantCheck((a + b)! = (a + b))$, $MutantCheck((a + b)! = (a/ b))$, and $MutantCheck((a + b)! = (a\#b))$. In the fourth column, each example consists of two parts. The bottom part in bold font denotes some snippet of an FDM specification, which matches the left part of mutation rules. Correspondingly, we instrument the generated MKC immediately before the bold text. Due to the limited space, we only present one possible mutation for each operator in column 4. If the conditional statement within $MutantCheck()$ holds, the specification and its mutant can be differentiated. As an example for the operator AOR, to satisfy the constraint we can generate some values for $a$ and $b$ (e.g., $a=1, b=3$) that can differentiate the statement “$c = a - b$.”

Based on the rules introduced in Table 1, Listing 1 presents an FDM excerpt of the e1000 network adapter instrumented MKCs. To facilitate the illustration, the names of some registers and macros are modified here. In this example, there are three MKCs (lines 15, 16, and 21) generated by the operators LCR, LCR, and UOI, respectively. Note that the function $MutantCheck()$ has two parameters, where the first one is a conditional statement denoting the MKC and the second one denotes the indices of mutants. By default, the index of a mutant is 0, if the second parameter is not specified.
Therefore, the test cases generated from FDMs can be directly used to check the conformance between FDMs and virtual/silicon devices via interface registers. Definition 3.1 and Definition 3.2 give the formal definition of test cases generated from FDMs.

**Definition 3.1.** A test case \( \tau \) generated from an FDM is a sequence of device requests in the form of \( \tau = req_0 \rightarrow req_1 \rightarrow \ldots \rightarrow req_n \) \((n \geq 0)\). Each device request \( req_i \) in \( \tau \) is a triple \((accType, regAddr, val)\) indicating an interface register access operation, where \( accType \in \{R,W\} \) denotes read or write type of \( req_i \), \( regAddr \) denotes the target register address, and \( val \) denotes the value going to be written to \( regAddr \) when \( accType = W \). The length of \( \tau \) is \(|\tau| = n + 1\).

For example, the device request sequence \( \tau = \{(W,0x8,0x0) \rightarrow (W,0x3818,0x1010101) \rightarrow (W,0x10,0x0) \rightarrow (W,0xb8,0x0) \rightarrow (W,0xc4,0x0) \rightarrow (W,0x400,0x2020202)\} \) is a generated test case for some FDM, which has a length of 6. Note that \( \tau \) is in an abstract form, which is device independent.

**Definition 3.2.** Let \( \tau = req_0 \rightarrow req_1 \rightarrow \ldots \rightarrow req_n \) \((n \geq 0)\) be a test case. A test segment \( ts_i \) of \( \tau \) is a sub-sequence of \( \tau \), where \(|ts_i| \leq |\tau|\). A continuous test segment sequence of \( \tau \) in the form of \( ts_0 \rightarrow ts_1 \rightarrow \ldots \rightarrow ts_k \) \((k \leq n)\) has the same device request sequence as \( \tau \) such that \( \sum_{i=0}^{k} |ts_i| = n + 1 \).

Let \( ts_0 = (W,0x8,0x0) \rightarrow (W,0x3818,0x1010101) \), \( ts_1 = (W,0x10,0x0) \rightarrow (W,0xb8,0x0) \), and \( ts_2 = (W,0xc4,0x0) \rightarrow (W,0x400,0x2020202) \) be three test segments of \( \tau \) with a length of 2, respectively. \( \tau \) can be considered as a continuous sequence of \( ts_0, ts_1 \), and \( ts_2 \), i.e., \( \tau = ts_0 \rightarrow ts_1 \rightarrow ts_2 \).

To enable device conformance checking, when running an instrumented FDM with a given test case, for each device request we need to record the corresponding runtime information including the device state update and the set of MKCs traversed during the test execution. Definition 3.3 formalizes such information that needs to be collected.

**Definition 3.3.** Let \( S \) be the set of interface register states of a virtual/silicon device \( d \), where each state denotes specific value assignments of its interface registers. Let \( s_0 \in S \) be an initial state of \( d \). The trace of \( d \) based on \( s_0 \) and \( \tau \) is a sequence \( \rho = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n \rightarrow s_{n+1} \), where \( s_{n+1} \) denotes the device state after executing \( req_n \) from \( s_n \) and \( \mu_i \) denotes the MKC triggered by \( req_i \).

To generate a device request to trigger one MKC \( m \) from current state \( s_i \), we need to figure out a test segment \( ts_i = req_0 \rightarrow req_1 \rightarrow \ldots \rightarrow req_k \) such that we can get \( m = \bigcup_{j=0}^{k} \mu_j \), where \( s_i \rightarrow (req_0, \mu_0) \rightarrow s_{i+1} \rightarrow (req_1, \mu_1) \rightarrow \ldots \rightarrow (req_k, \mu_k) \rightarrow s_{i+k+1} \) is the trace segment. We say that \( ts_i \) can trigger \( m \) from \( s_i \). To enable the investigation of complex functional scenarios, Definition 3.4 defines a new test generation method involving the continuous killing of multiple mutants within a test case execution.

**Definition 3.4.** Let \( IM \) be the set of instrumented MKCs of an FDM \( f \), and let \( mc = \langle m_{i_0}, m_{i_1}, \ldots, m_{i_k} \rangle \) \((m_{i_j} \in IM, k \geq 0, 0 \leq j \leq k)\) be a combination of \( k + 1 \) MKCs. A test case \( \tau \) can cover \( mc \) if there exists an initial device state \( s_0 \) and a continuous test segment sequence \( \tau = ts_0 \rightarrow ts_1 \rightarrow \ldots \rightarrow ts_k \) for \( f \) such that \( ts_i \) can trigger \( m_{i_j} \) within the trace of \( f \) based on \( s_0 \) and \( \tau \). We use \( Mx \) to denote the test cases generated to cover all the combinations of \( x \) MKCs in \( IM \).
According to [5], each FDM iteration needs to deal with two functions runInterfaceFunction() and runDevice(), where runInterfaceFunction() specifies interface register access behaviors and runDevice() updates the device state accordingly. Since these two functions are the main parts to describe device behaviors, our approach instruments them with MKCs for test case generation. As an example, Listing 1 shows the instrumented functions, which will be used in explaining our mutant-driven test case generation approach. The function terminateState() in line 11 is a special function defined in our modified KLEE, which acts like a barrier for the symbolic search. If at the end of this barrier none of the execution paths encounters an uncovered MKC, the whole-loop will be unrolled again to search for uncovered MKCs with longer execution paths. Otherwise, the terminateState() function will select one execution path with specific strategy (see lines 27-34 in Algorithm 2) for device request generation. If the while-loop is unrolled twice and still cannot find any new uncovered MKC, terminateState() will randomly generate one device request from the explored execution paths. Then, terminateState() will feed the generated device request to the corresponding device.

```c
int main(int argc, char *argv[])
{
    struct DevState dev; int cnt=0;
    getDevState(&dev);
    while(cnt<2)
    {
        uint32_t acc_val; uint64_t addr; cnt++;
        klee_make_symbolic(acc_val, sizeof(acc_val), "accessType");
        klee_make_symbolic(addr, sizeof(addr), "regAddr");
        klee_make_symbolic(val, sizeof(val), "value");
        runInterfaceFunction(&dev, acc_val, addr);
        runDevice(&dev);
        terminateState();
    }
    ...
}
```

Listing 2. FDM Harness for Test Case Generation

Figure 4 shows an example of M1 test generation for the FDM modeled in both Listing 1 and Listing 2. For the ease of explanation, we only present the programming constructs that correspond to the plaintext in Listing 1. In this figure, each FDM programming construct (node) is labeled with an ID, and we use $m_i$ to indicate the $i^{th}$ MKC. Note that a conditional statement wrapped in mutantCheck() is only for test case generation. During symbolic execution, if an execution path triggering an uncovered MKC is selected for device request generation, the MKC will be incorporated in the corresponding path constraint by KLEE. Otherwise, the mutantCheck() will be skipped.

Figure 4 shows a simplified test case generation process for MKCs $m_1$, $m_2$ and $m_3$. For simplicity, we assume that each generated device request kills only one MKC. In this case, since each test case involves only one device request, the test case generation process only needs to invoke the symbolic execution of the harness for three times. We use different colors to indicate different symbolic executions. At the beginning of M1 test case generation, our approach parses the FDM to figure out the set $mutComCov = \{ <m_1>, <m_2>, <m_3> \}$ of all combinations of MKCs. Our approach uses a set $tList$ to save the execution constraints for test case generation. In the first symbolic execution (in red color), when reaching node 2, the current execution path state $\rho$ forks two execution paths (i.e., $\rho = 1 \rightarrow 2 \rightarrow \ldots$, $\rho = 1 \rightarrow 2 \rightarrow 3$). Since we omit the part between node 2 and node 8, in this example $p_1$ will not be discussed. Assuming that $\rho$ has a higher priority, $p_1$ will be saved temporarily. When our symbolic execution hits the node 3, $\rho$ will be forked again into two execution paths (i.e., $\rho = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, $\rho = 1 \rightarrow 2 \rightarrow 3 \rightarrow 7$). Assuming that $\rho$ has a higher priority, $p_2$ will also be saved for future analysis. When the search of $\rho$ hits node 4, it will check mutComCov to find whether $m_1$ has been triggered. Since $<m_1>$ is in mutComCov, our approach will update $tList = \{ m_1 : A \land B \land D \}$ assuming that $A \land B \land D$ is satisfiable. After that, $\rho$ will hit $m_2$ at node 5. By checking mutComCov, we will update $tList = \{ m_1 : A \land B \land D, m_2 : A \land B \land G \}$ assuming that $A \land B \land G$ is satisfiable. When $\rho$ hits node 6, it will fork two execution paths (i.e., $\rho = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 8$, $\rho = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow \ldots$). Assuming that $\rho$ has a higher priority, $p_3$ will not be discussed due to lack of successive nodes. Thereafter, $\rho$ will hit node 8, where it will be suspended temporarily. After that $p_2$ will be explored. When $\rho$ hits node 7, after checking mutComCov, $tList$ will be updated to $\{ m_1 : A \land B \land D, m_2 : A \land B \land G, m_3 : A \land C \land E \}$. When all the active execution paths finish the exploration, the terminateState() will terminate the first symbolic execution and select $m_1$ from $tList$ to generate a test case and send the device request to its corresponding device. Meanwhile, $tList$ will be cleared, and mutComCov will become $\{ <m_2>, <m_3> \}$. Similarly, the second symbolic execution explores the harness function in the same way. The only difference is that $m_1 : A \land B \land D$ will not be added to $tList$, since $<m_1>$ is not in mutComCov. In this case, a device request for $m_2$ will be generated. In the last round, $m_3$ will be handled in the same manner.

### 3.2.4 Implementation of Test Case Generation

Instead of giving the details of the special functions (e.g., terminateState()) newly defined in KLEE, Algorithm 1 details our test case generation approach as a whole from a global view. At the beginning, line 2 parses the given FDM to compute the set of all the instrumented MKCs. Line 3 resets the device. Since the generation of device requests highly depends on current device states, inevitably the symbolic execution may get stuck at local search without generating device requests to trigger new MKCs. Therefore, we introduce two random search mechanisms to avoid these scenarios. Our approach allows random device requests if no effective device requests can be generated. If there exist $rstBound$ (default value is 5) continuous random device requests, our approach will reset the device once (lines 32-34). If there are $rdnRunCnt$ (default value is 1000) continuous random device requests generated after an effective device request, the whole test case generation process will abort (line 6, lines 26-
Algorithm 1: Our Test Case Generation Algorithm for $M_k$

Input: i) $fdm$, an instrumented FDM ;
ii) $dev$, corresponding virtual (silicon) device;
iii) $k$, # of mutants in a combination of $M_k$;
iv) $rndBound$, # of max random tries for killing a new mutant;
v) $rstBound$, # of continuous random tries for reset;

Output: Trace (test case) set $TCS$ for killing $k$-mutation-combinations
1 TestCaseGeneration($fdm$, $dev$, $k$, $rndBound$, $rstBound$) begin
2     $mutSet = ParseFDM(fdm)$;
3     $ResresetDev(dev)$;
4     $rndRunCnt=0$, $reqCnt=0$;
5     $reqEventSeq=Clea();$
6     while $rndRunCnt < rndBound$ do
7         $curSta = GetRegState(dev)$;
8         $executionState = ExploreState($dev, $curSta$, $mutSet$, $k$);
9         if $executionState != NULL$ then
10            $mud = executionState.idMKC$;
11            $req = RequestGen(executionState)$;
12            $SendRequestToDevice(req)$;
13            $nxtSta = GetRegState(dev)$;
14            $killMutSeq.Append(idMKC)$;
15            if $reqCnt == k$ then
16                $reqCnt=0$;
17                $mutComCov.Add(killedMutSeq)$;
18                $TCS.Add(reqEventSeq)$;
19                $reqEventSeq.Clear();$
20                $killMutSeq.Clear();$
21            end
22        else
23            $reqEventSeq.Append($curSta$, $nxtSta$, $mudD >)$;
24        end
25        $rndRunCnt += 1$;
26    end
27    end
28    else
29        $RandomRun(dev)$;
30        $rndRunCnt += 1$;
31    end
32    if $rndRunCnt % rstBound == 0$ then
33        $ResetDevice(dev)$;
34    end
35 return $TCS$;
36 end

31). Line 4 initializes the values of counters for random runs and generated device requests. Since $M_k$ test cases contain at least $k$ request events, we use $reqEventSeq$ to save the sequence for a test case. Line 5 resets $reqEventSeq$ for a new test case generation. Note that an $M_k$ test case may involve multiple device requests. To enable conformance checking, our approach saves all the information of a generated device request in the form of a request event sequence, which is denoted by $req$, respectively. We call such a record an event of a generated device request in the form $(< req, curSta, nxtSta, mud>)$.

lines 6-10 will ignore the MKC. Note that lines 6-10 iteratively compare the possible uncovered MKCs for current device request generation. As shown in lines 1-2, we use $availMKC$, $score$, $selExecState$, and $tList$ to denote the MKCs that need to be checked, the score of each MKC based on the statistics of $mutComCov$, the selected execution state triggering an uncovered MKC, and the candidate execution states for test case generation, respectively. Lines 4-12 figure out possible uncovered MKCs for current device request generation. If the device request is not the last one in an $M_k$ test case, all the MKCs will be explored. Otherwise, if the combination of $killMutSeq$ and an MKC has been covered in $mutComCov$, the selected execution state triggering an uncovered MKC, and the candidate execution states for test case generation, respectively.

Algorithm 2 details the symbolic execution-based exploration process for a best candidate to trigger a new MKC. As shown in lines 1-2, we use $availMKC$, $score$, $selExecState$, and $tList$ to denote the MKCs that need to be checked, the score of each MKC based on the statistics of $mutComCov$, the selected execution state triggering an uncovered MKC, and the candidate execution states for test case generation, respectively. Lines 4-12 figure out possible uncovered MKCs for current device request generation. If the device request is not the last one in an $M_k$ test case, all the MKCs will be explored. Otherwise, if the combination of $killMutSeq$ and an MKC has been covered in $mutComCov$, the selected execution state triggering an uncovered MKC, and the candidate execution states for test case generation, respectively. Lines 4-12 figure out possible uncovered MKCs for current device request generation. If the device request is not the last one in an $M_k$ test case, all the MKCs will be explored. Otherwise, if the combination of $killMutSeq$ and an MKC has been covered in $mutComCov$, the selected execution state triggering an uncovered MKC, and the candidate execution states for test case generation, respectively. Lines 4-12 figure out possible uncovered MKCs for current device request generation. If the device request is not the last one in an $M_k$ test case, all the MKCs will be explored. Otherwise, if the combination of $killMutSeq$ and an MKC has been covered in $mutComCov$, the selected execution state triggering an uncovered MKC, and the candidate execution states for test case generation, respectively. Lines 4-12 figure out possible uncovered MKCs for current device request generation. If the device request is not the last one in an $M_k$ test case, all the MKCs will be explored. Otherwise, if the combination of $killMutSeq$ and an MKC has been covered in $mutComCov$, the selected execution state triggering an uncovered MKC, and the candidate execution states for test case generation, respectively.
execution state. As shown at line 18, if an MKC collected in line 16 is in availMKC, an execution state indicated by es together with the MKC will be considered as a candidate. If this MKC combined with the path constraints of es is satisfiable, line 22 will add it into tList. Lines 27-31 update the scores of MKCs, and lines 32-34 select one execution state from tList with the lowest MKC score. Finally, line 35 returns the selected state if it exists.

### 3.3 Symbolic Execution-based Conformance Checking

Our approach conducts the conformance checking between FDMs and virtual/silicon devices based on the symbolic execution of instrumented FDMs (see Section 3.1) using the traces collected from virtual/silicon devices. Since the usages of interface registers are the same across different layers of device design (i.e., FDMs, virtual/silicon devices), the collected traces from virtual/silicon devices (see Algorithm 1) can be directly applied to FDMs.

According to Algorithm 1, each collected trace is a sequence of events, where each event is in the form of < req, curSta, nxtSta, muID > indicating the incoming device request, current state, next state and the ID of investigating mutant, respectively. Assume that the state of an FDM f is initialized with curSta. When we symbolically execute f with req, if the corresponding device implementation dev conforms to f, the next state of f should be consistent with nxtSta. However, if the next state of f is consistent with nxtSta, it does not mean that the implementation is correctly designed. As an example shown in Figure 1, if the inputs of a and b are all 0, the next states of both given specification and implementation are consistent, which results in a false positive during conformance checking. To avoid such kinds of false positives, our conformance checking requires that the MKC with index muID should also be checked during the symbolic execution.

#### 3.3.1 Formal Definitions

Both FDMs and virtual/silicon devices consist of two kinds of registers: i) interface registers that explicitly reflect the interactions with other software/hardware components; and ii) internal registers used for internal calculations that are not observable. Assume that Rf and Rn are the set of interface registers and internal registers, respectively. Definition 3.5 and Definition 3.6 present the formal definitions on checked states of devices and instrumented FDMs, respectively.

**Definition 3.5.** After a device request req, the state of a device is denoted as \( S = \{ S_I, S_N \} \), where \( S_I \) and \( S_N \) denote the assignments to \( R_I \) and \( R_N \) due to the execution of req, respectively. The checked state of a device is denoted as \( S_{\text{conf}} = \{ S_I, S_N \} \) where \( ID_p \) is the index of investigating MKC associated with req.

**Definition 3.6.** Let \( p \) be a symbolic execution path of an instrumented FDM \( fd \) when dealing with a device request. At the end of symbolic execution of \( p \), the state of \( fd \) is denoted as \( F = \{ F_I, F_N \} \), where \( F_I \) and \( F_N \) are the final assignments to \( R_I \) and \( R_N \), respectively. The checked state of \( fd \) under \( p \) is denoted as \( CF_p = \{ F_I, F_N \} \), where \( IDS_p \) is the index set of all MKCs triggered along with \( p \).

In symbolic execution, the register values of concrete FDM/device states are all concrete, while the registers are assigned with symbolic values for symbolic FDM/device states. As an abstraction, a symbolic FDM/device state can be treated as a set of concrete FDM/device states. To facilitate the definition, the states of FDMs and virtual/silicon devices are all considered as symbolic states. Similar to the conformance checking approaches proposed in [31], [5], we use \( \text{Sym}(S) \) and \( \text{Sym}(F) \) to denote the sets of concrete states for \( S \) and \( F \), respectively. Note that unlike existing approaches, our method takes mutant-killing information into account. Definition 3.7 and Definition 3.8 define the conformance between FDMs and device implementations considering only one device request.

**Definition 3.7.** A checked state of virtual/silicon devices \( CS = \{ S_I, S_N \} \) and a checked state of corresponding FDM \( CF_p = \{ F_I, F_N \} \) are consistent to each other if both of \( \text{Sym}(\{ S_I, S_N \}) \cap \text{Sym}(\{ F_I, F_N \}) \neq \emptyset \) and \( ID_p \in IDS_p \) are satisfied.

**Definition 3.8.** Let \( fd \) be an instrumented FDM and \( d \) be its implementation. After synchronizing the state of \( fd \) with the state of \( d \), both designs are executed with a same device request req. Let \( CS \) be the checked state of \( d \) after executing req. Let \( P \) be the set of all possible execution paths of \( fd \) during the symbolic execution of req. For the device request req, d conforms to \( fd \) if there exist \( p \in P \) such that \( CS \) is consistent to \( CF_p \).

#### 3.3.2 Implementations

Unlike the harness presented in Listing 2 that is used for test case generation, Listing 3 presents the FDM harness for conformance checking. By replacing the condition statements in the while-loop and switch-case constructs with a symbolic condition \( \text{choice()} \), our approach can reflect the real hardware behaviors by executing functions \( \text{runInterfaceFunction()} \) and \( \text{runDevice()} \) (see details in Listing 1) in a non-deterministic manner.

```c
1. int main(int argc, char *argv[]){
2.   struct DevState dev;
3.   uint32_t acc,val;
4.   uint64_t addr;
5.   while(choice()){
6.     switch(choice()){
7.       case 0 :
8.         runInterfaceFunction(ddev, acc, val, addr);
9.         break;
10.      case 1:
11.         runDevice(ddev);
12.         ...}
13.    ...}
14.    ...}
15. static inline int choice();
16.    ...}
17. make_symbolic(&i, sizeof(i), "choice");
18.    return i;}
```

Listing 3. Harness of FDM for Conformance Checking

According to Definition 3.8, our approach can check the conformance between FDMs and devices under a given device request. While analyzing traces collected from our mutant-driven testing, we check each request in a trace one by one. If an inconsistency is detected between the device and its FDM under a request, our approach will not be terminated immediately. This is because we use an FDM-device synchronization mechanism to perform the test generation and execution. In this way, the conformance between the FDM and device for each device request is independent.
Algorithm 3 details our conformance checking approach by running a collected device trace on the corresponding instrumented FDM. By parsing the trace file, line 4 figures out all the collected device information for a given device request. To reduce symbolic execution complexities, our approach adopts the method proposed in [5], [31], which synchronizes the FDM state to its corresponding device state after each device request. Based on a new device state according to curSta and a given device request req, line 5 fully explores all the possible execution paths and save them in a set execPaths. Then, lines 6-11 iteratively check whether there exists an execution path whose checked state is consistent with < mutID, nxtSta > according to Definition 3.7. If the device does not conform to the FDM for req as shown in line 12, line 13 will record the inconsistency.

Algorithm 3: Our Conformance Checking Approach

Input: i) fdm, an instrumented FDM; 
ii) trace, a trace generated by Algorithm 1;

1 ConformanceChecking(fdm, trace) begin
2 for i from 0 to trace.size()-1 do
3     mis = TRUE;
4     < req.curSta,nxtSta,mulID > = trace.At(i);
5     execPaths = SymbolicExecute(fdm, curSta, req);
6     while (execPaths.isEmpty()) do
7         ep = execPaths.Remove(0);
8         if (CheckStMut(ep, nxtSta, muID) then
9             mis = FALSE; break;
10       end
11     end
12     if mis == TRUE then
13         RecordMisInfo(trace.At(i));
14     end
15 end
16 end

4 PERFORMANCE EVALUATION

To evaluate the effectiveness of our approach, we conducted the experiments with two industrial network adapters (i.e., e1000 Gigabit NIC and eepro100 Megabit NIC developed by Intel). These two adapters have both virtual and silicon versions, where their virtual prototypes can be obtained from the virtual machine QEMU (version 0.15). Note that e1000 is the default network adapter of QEMU. We modified the open-source C mutation testing tool Milu [37], which can parse FDM specifications and instrument MKCs automatically based on the given mutation operators. We also modified the symbolic execution tool KLEE (version 1.4.0) and incorporated our test generation, test execution and trace comparison approaches in it to enable the conformance checking between FDM specifications and device implementations. Note that KLEE can record the line number information of symbolically executed code. It can be used to compute the code and mutant coverage of FDMs. To collect the coverage information of virtual devices using the generated test cases, we use the GCC’s coverage testing tool GCOV and LCOV. All the experiments were conducted on an Ubuntu Desktop (version 12.04) with 3.2GHz AMD processors and 16GB RAM.

Table 2 presents the experimental settings for the two network adapters. Column 1 gives the name of the design. Based on Intel developers’ manuals [38], [39], we constructed the FDMs for the two adapters. Columns 2-3 present the Lines of Code (LoC) information for the FDMs and their instrumented versions, respectively, while column 4 gives the number of generated MKCs based on our proposed mutation operators. For the ease of comparison, we do consider header files when calculating LoC information of both FDMs and virtual prototypes. Note that FDMs are high-level abstractions of devices that only take partial interface registers of the investigated network adapters into account. In this experiment, we only investigate 26 interface registers (20 non-reserved and 6 reserved ) for the e1000, and 13 interface registers (12 non-reserved and 1 reserved) for eepro100. All these registers are among the most frequently used registers in their virtual prototype counterparts. Column 5 presents the LoC information for the virtual prototypes obtained from QEMU. Instead of monitoring all the register updates, when collecting execution traces from virtual and silicon devices, we only record a subset of interface registers which are relevant to the selected registers used in modeling FDMs. Column 6 shows the total address range of such monitored interface registers.

As shown in Figure 5, our experiment involves three different design layers: FDM specifications, virtual prototypes, and silicon devices. The arcs on the left side indicate the implementation validation using the test cases generated by our mutation-driven approach, while the arcs on the right side denote the conformance checking by symbolically executing the implementation traces stimulated by our generated test cases on FDM specifications. Our experiment tries to answer the following three questions.

RQ1: How is the quality of the test cases generated by our proposed mutation-driven approach?
RQ2: Can our approach really reduce the overall conformance checking time, thus shortening the time-to-market?
RQ3: How effective is our approach in detecting the inconsistencies between FDM specifications and device implementations?

4.1 Adequacy Analysis of Generated Test Cases

To demonstrate the quality of generated test cases, we investigated all the mutation operators as proposed in Section 3.1 for the test case generation of the two network adapters. For each FDM instrumented with automatically generated MKCs, we generated

![Fig. 5. Conformance checking relations of three different design layers](image-url)
a set of test cases for the validation of its corresponding virtual prototype. Note that we did not check the testing adequacy for silicon devices because of their limited observability.

### Table 3

| Mutant Killing Information of Test Case Generation |
|---------------------------------|---------------|-----------------|---------------|-----------------|---------------|-----------------|-----------------|
| Methods                        | e1000 FDM     | Total Succ.     | Total Fail.    | ego100 FDM     | Total Succ.     | Total Fail.    |
| M1                             | 115           | 112             | 3              | 14641          | 7074           | 7567           |
| M2                             | 13225         | 12700           | 525            | 95.3           | 47.7           | 1m20s          |

Table 3 presents the mutant killing information for the test generation from FDM specifications. Column 1 presents different test case generation strategies. According to Definition 3.4, we use the notation $M_x$ to denote the strategy where each generated test case involves the killing of a combination of $x$ mutants. Column 2 has three sub-columns, which indicate the mutant killing information for the e1000 FDM. The first sub-column denotes the number of all generated mutant combinations that are used for test case generation. Note that the number of mutation combinations increases exponentially along with the increase of $x$. For example, the number of M2's combinations is the square of the number of M1's combination. To achieve a reasonable conformance checking time, this experiment does not check the case where $x > 2$. Since there exist mutants that contradict each other, the second and third sub-columns present the number of mutant combinations that are successfully and unsuccessfully killed using our test case generation approach, respectively. For each successfully killed mutant combination, our approach generates one test case for the validation of virtual/silicon devices. Similarly, the third column shows the mutant-killing information of the test case generation for ego100 FDM.

**Virtual Device Bugs Detected.** For the e1000 virtual device, when applying the 12700 test cases generated by M2, we detected device bugs that have not been identified by the command-based test cases used in [5], [31]. We found that the guest Linux operating system running on QEMU often got stuck with infinite loops. We analyzed the virtual prototype’s source code based on the reports in [40], [41], and figured out the two bugs: i) due to the missing of a break statement, the while-loop for data transmission cannot terminate; and ii) if the value of the variable “bytes” becomes 0 without proper checking, the condition of some while-loop for processing transmit descriptors will always be true. For the completeness of the conformance checking, all the following experiments are based on the e1000 virtual device with the two detected bugs fixed.

### Table 4

<table>
<thead>
<tr>
<th>Devices</th>
<th>Test Cases</th>
<th>Time</th>
<th>Line Cov. (%)</th>
<th>Func. Cov. (%)</th>
<th>Branch Cov. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1000</td>
<td>Com [5]</td>
<td>25m</td>
<td>79.9</td>
<td>81.4</td>
<td>55.8</td>
</tr>
<tr>
<td>M1</td>
<td>5m</td>
<td>75.5</td>
<td>90.7</td>
<td>47.5</td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td>6835m</td>
<td>80.0</td>
<td>95.3</td>
<td>66.0</td>
<td></td>
</tr>
<tr>
<td>M1+M2</td>
<td>6837m</td>
<td>88.0</td>
<td>95.3</td>
<td>66.0</td>
<td></td>
</tr>
<tr>
<td>M1+M2+Com</td>
<td>7620m</td>
<td>88.0</td>
<td>95.3</td>
<td>66.0</td>
<td></td>
</tr>
<tr>
<td>ego100</td>
<td>Com [5]</td>
<td>15m</td>
<td>68.8</td>
<td>73.8</td>
<td>42.2</td>
</tr>
<tr>
<td>M1</td>
<td>1m20s</td>
<td>71.2</td>
<td>73.8</td>
<td>47.3</td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td>1h20m</td>
<td>71.2</td>
<td>73.8</td>
<td>47.7</td>
<td></td>
</tr>
<tr>
<td>M1+M2</td>
<td>1h22m</td>
<td>71.2</td>
<td>73.8</td>
<td>47.7</td>
<td></td>
</tr>
<tr>
<td>M1+M2+Com</td>
<td>1h40m</td>
<td>71.2</td>
<td>73.8</td>
<td>47.7</td>
<td></td>
</tr>
</tbody>
</table>

To show the strength of our approach, we compared our approach with the work presented in [5], which runs virtual prototypes using a set of frequently-used network commands (e.g., *scp*, *ifconfig*) with the help of device drivers. Since the device requests in this case are derived from network commands rather than FDMs, the conformance checking method proposed in [5] lacks the controllability of error exploration, whereas our approach is more targeted. Note that during the early stage of computer system design when drivers are not ready, our approach is more suitable for testing purpose. Table 4 presents the testing results for the two virtual prototypes as indicated in column 1. Column 2 presents the methods used for test case generation, where Com denotes the command-based approach proposed in [5] and the notation “+” denotes the union of test case sets generated by different methods. Column 3 gives the overall testing time including both test case generation time and test execution time. Note that the test case generation time for Com is 0. The last three columns report the line coverage, function coverage and branch coverage for different test case sets, respectively.

From Table 4, we can observe that M1 spends much less testing time than Com. This is because Com executes more than one million device requests for each design, while M1 only generates and executes 112 device requests for e1000 and 83 device requests for ego100, though the majority of testing efforts involving M1 and M2 are spent on test case generation. However, from ego100, we can observe that M1 outperforms Com for better coverage in all listed categories. Even for e1000, M1 can achieve better function coverage than Com. Therefore, if the testing time is a major concern of validation under the time-to-market pressure, M1 could be a reasonable choice. Since M2 generates more test cases than M1 to cover various complicated mutation combinations (see Table 3), it can achieve better coverage results than M1. By comparing all the results of M2, M1+M2 and M1+M2+Com, we can infer that M2 achieves the highest coverage. In other words, M2 can explore more potential inconsistent scenarios than M1 and Com. Note that the two FDMs only consider a limited number of interface registers. If more registers are investigated in FDM modeling, our approach (M1 and M2) can obtain better coverage results than the ones presented in Table 4.

#### 4.2 Performance Analysis of Conformance Checking

Although M2 needs longer overall testing time than Com assuming that the commands are all collected in advance, it does not indicate that the conformance checking time of M2 is longer as well. When collecting execution traces of device implementations, each test case may involve at most 4 device requests. As an example for e1000 FDM, since there are 12700 test cases generated for conformance checking, the conformance checking will deal with $12700 \times 4 = 50800$ device requests at most. However, when adopting Com for conformance checking, the implementation traces collected from commands may be extremely long [5]. For example, the command *scp* for transferring a large file may consist of thousands of or millions of events depending on the file size. For the Com method presented in Table 4, the test cases of both NIC designs consists of more than one million events.

Figure 6 compares the conformance checking time for e1000 NIC and ego100 NIC based on the virtual/silicon device traces collected using different testing approaches, respectively. It can be observed that our approaches (i.e., M1, M2, M1 + M2) outperform Com by several orders of magnitude in terms of conformance checking time. As an example of ego100 NIC, the conformance checking time using Com is 99.95 hours, while our M1 + M2...
approach just needs 0.78 hours. The reason of this significant improvement is because our approach can generate a succinct set of shorter and well-targeted test cases, while the implementation traces generated by Com contains a large set of random but redundant device requests.

![Comparison of conformance checking time with different methods](a) Intel e1000 NIC (b) Intel eepro100 NIC

Fig. 6. Comparison of conformance checking time w/ different methods

To show the sufficiency of our conformance checking approach, Table 5 compares our approach with Com [5] in terms of mutant coverage (i.e., mutant killing ratio) and line coverage of FDMs. In this table, we investigated both traces collected from virtual/silicon devices for the two NIC designs. Based on both results of Table 5 and Figure 6, we can find that our approach can achieve better mutant and line coverages than Com with much fewer conformance checking efforts. This is mainly because the test case generation used in our approach is well-directed by the mutations.

### TABLE 5

**Mutant and Line Coverages of FDMs**

<table>
<thead>
<tr>
<th>Mutation Methods</th>
<th>e1000-VP</th>
<th>e1000-SD</th>
<th>eepro100-VP</th>
<th>eepro100-SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Com [5]</td>
<td>39.13/63.40</td>
<td>41.74/58.39</td>
<td>64.36/76.59</td>
<td>63.64/83.33</td>
</tr>
<tr>
<td>M1</td>
<td>96.52/97.74</td>
<td>93.04/90.94</td>
<td>66.94/84.92</td>
<td>67.77/84.92</td>
</tr>
<tr>
<td>M2</td>
<td>96.52/98.39</td>
<td>94.78/92.83</td>
<td>66.94/84.92</td>
<td>67.77/84.92</td>
</tr>
<tr>
<td>M1+M2</td>
<td>96.52/98.39</td>
<td>94.78/92.83</td>
<td>66.94/84.92</td>
<td>67.77/84.92</td>
</tr>
</tbody>
</table>

#### 4.3 Identified Inconsistencies

To show the effectiveness of our approach in detecting inconsistencies, we stimulated both virtual and silicon devices of the two NIC designs using the test cases generated by M1 and M2, respectively. We identified inconsistencies between FDMs and virtual/silicon devices by symbolically executing the instrumented FDMs with the collected traces. Table 6 presents the identified inconsistency results, where the notation M1/M2/Com denotes the number of inconsistencies identified by M1, M2, and Com, respectively. From this table, we can find that all the three methods can identify the same number of distinct inconsistencies. We further checked the inconsistency records, and found that the sets of inconsistencies identified by the three methods are the same. It means that with much less conformance checking efforts (see Figure 6) our approach can identify the same inconsistencies as Com [5].

Our approach assumes that the FDM is the golden reference model for device specification, i.e., the FDM covers all allowed behaviors of the device and any inconsistency with the FDM is a violation of the device specification. So the insufficiency in the modeling of FDM is not within our consideration. Towards this end, our conformance checking will not cause false positives, i.e., that an allowed device behavior is classified as not allowed. On the other hand, our conformance checking may have false negatives, i.e., that not all inconsistencies with the FDM are detected. This is because our approach is testing based by nature; therefore, does not entail complete coverage of all device behaviors.

Table 7 presents an inconsistency example for e1000 that is caused by a device request that tries to write the register MDIC at address 0x20 with a value 0x14200000. Our framework can detect the inconsistency where the register ICR at address 0xC0 in virtual device has a value of 0x80002000, while the ICR register in the FDM specification has a value of 0x0. According to the specification, if the 29th bit of MDIC is set to 1, it will cause an interrupt indicated by the 9th bit of ICR. Moreover, the highest bit of ICR should have a value of 0, since it is a reserved bit in the specification. When the input for MDIC is 0x14200000, its 29th bit equals 0. In this case, no interrupt will be invoked. However, we can find that in the virtual device the 9th bit of ICR equals 1. Therefore, based on this inconsistency our approach can find an error from the virtual device. Table 7 only shows the scenario with a specific write value. In fact, there exist a large number of errors of the same type as the one shown in Table 7. When adopting the Com [5] method, 1859224 device requests were generated, and 0.17% of them can be used to detect errors of this type. However, among all the 25512 device requests generated by our M1+M2 method, 8.47% of them can detect such kind of errors. We can find that our approach has a higher chance to trigger the inconsistencies between specifications and implementations. This is mainly because our test cases are more targeted under the guidance of MKCs. It is important to note that, although the generated device requests by our approach M1+M2 are far fewer than the ones generated by Com, our approach can detect same types of inconsistencies as Com as shown in Table 6.

### TABLE 6

**Inconsistencies Identified from Virtual/Silicon Devices**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>e1000 Bug #</th>
<th>eepro100 Bug #</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>Update reserved SD registers</td>
<td>3/3</td>
<td>0/0</td>
</tr>
<tr>
<td>E2</td>
<td>Update reserved VP registers</td>
<td>0/0</td>
<td>1/1</td>
</tr>
<tr>
<td>E3</td>
<td>Generate unnecessary interrupts</td>
<td>0/0</td>
<td>1/1</td>
</tr>
<tr>
<td>E4</td>
<td>Fail to update necessary registers</td>
<td>0/0</td>
<td>1/1</td>
</tr>
<tr>
<td>E5</td>
<td>Write incorrect values to registers</td>
<td>3/3</td>
<td>0/0</td>
</tr>
</tbody>
</table>

### TABLE 7

**An Example of Register Value Inconsistency**

<table>
<thead>
<tr>
<th>Request</th>
<th>Write MDIC (addr=0x020) with value: 0x14200000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inconsistent Info.</td>
<td>Value in Virtual Device</td>
</tr>
<tr>
<td>ICR (addr=0x0C)</td>
<td>0x80000200</td>
</tr>
</tbody>
</table>

#### 5 Conclusions and Future Work

This paper presented a mutation-driven conformance checking framework that enables the effective exploration of inconsistencies between device specifications and implementations. Based on our proposed mutation operators and cooperative symbolic execution method, our framework can automatically generate effective test cases to check various error-prone functional scenarios for the implementations (i.e., virtual/silicon devices). By symbolically executing virtual/silicon device traces triggered by the test cases.
generated from the specifications instrumented with weak MKCs, our approach can report the validation adequacy information as well as the bugs caused by inconsistent implementations. Experimental results using two industrial network adapters show that our approach can not only identify real bugs and inconsistencies in both virtual devices excerpted from QEMU and their silicon counterparts, but also generate a succinct set of tests that achieves better coverage than state-of-the-art methods.

Since mutation operators play an important role in the effectiveness and efficiency of mutation testing, in the future we plan to investigate more new mutation operators to enhance the capability of our approach in detecting more inconsistencies between FDMs and virtual/silicon devices. Moreover, to improve the overall conformance checking performance, how to reduce the complexity of symbolic execution-based test case generation and how to optimize mutation reduction strategies [43] while satisfying the checking sufficiency are also interesting topics that are worthy of further study.

ACKNOWLEDGMENTS
This work received financial support in part from National Key Research and Development Program of China (Grant #: 2018YFB2101300), Natural Science Foundation of China (Grant #: 61872147), and National Science Foundation (Grant #: 1908571). Mingsong Chen is the corresponding author.

REFERENCES


**Halifeng Gu** received the BE degree from the Department of Computer Science and Technology, Sichuan Normal University, Chengdu, China, in 2013. He is currently working toward the PhD degree in the Department of Embedded Software and System, East China Normal University, Shanghai, China. His research interests include the area of hardware/software co-validation, symbolic execution, statistical model checking, and software testing.

**Jianning Zhang** received the BE degree from the Software Engineering Institute, East China Normal University, in 2018. He is currently working toward the PhD degree in the Department of Embedded Software and System, East China Normal University, Shanghai, China. His research interests include the area of computer architecture, design automation of embedded systems, and software engineering.

**Li Lei** received the Ph.D. degree in Computer Science from Portland State University in 2015. He is currently a Research Scientist at Security and Privacy Research, Intel Labs. His research interests are primarily in the areas of computer security, system design and validation, and software engineering. He is currently focusing on building secure and reliable computer systems with Intel hardware supports as well as semi-formal methods.

**Mingsong Chen** (M’08–SM’17) received the B.S. and M.E. degrees from Department of Computer Science and Technology, Nanjing University, Nanjing, China, in 2003 and 2006 respectively, and the Ph.D. degree in Computer Engineering from the University of Florida, Gainesville, in 2010. He is currently a Professor with the Software Engineering Institute at East China Normal University. His research interests are in the area of cloud computing, design automation of cyber-physical systems, parallel and distributed systems, and formal verification techniques. He is an Associate Editor of IET Computers & Digital Techniques, and Journal of Circuits, Systems, and Computers.

**Tongquan Wei** (S’06–M’11) received his Ph.D. degree in Electrical Engineering from Michigan Technological University in 2009. He is currently an Associate Professor in the School of Computer Science and Technology at the East China Normal University. His research interests are in the areas of green and reliable embedded computing, cyber-physical systems, parallel and distributed systems, and cloud computing. He serves as a Regional Editor for Journal of Circuits, Systems, and Computers since 2012. He also served as Guest Editors for several special sections of IEEE TII and ACM TECS.

**Fei Xie** received the Ph.D. degree in Computer Science from the University of Texas at Austin in 2004. He is currently a professor in the Department of Computer Science, Portland State University. His research interests are primarily in the areas of embedded systems, software engineering, and formal methods. He is particularly interested in development of formal method based techniques and tools for building safe, secure, and reliable software and embedded systems.