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-

- *Haifeng Gu, Jianning Zhang, Mingsong Chen and Tongquan Wei are with the MoE Engineering Research Center of Software/Hardware Co-design Technology and Application, East China Normal University, Shanghai, 200062, China (email: {hfgu, jnzhang, mschen}@sei.ecnu.edu.cn, tqwei@cs.ecnu.edu.cn). Mingsong Chen is also with the Shanghai Institute of Intelligent Science and Technology, Tongji University. Li Lei is with Intel Labs, Hillsboro, OR 97124, USA (email: li.lei@intel.com). Fei Xie is with the Department of Computer Science, Portland State University, Portland, OR 97207, USA (email: xie@pdx.edu).*

Killing Constraints (MKCs) within a single FDM to automatically guide the test case generation by symbolic execution. This paper makes the **following three major contributions**:

1) We propose a set of mutation operators for the FDMs generated from xSystemRDL specifications. The mutants generated from these operators enable the modeling of implementation inconsistencies that lead to erroneous system behaviors.
2) Based on the weak MKCs instrumented in FDMs, we develop a cooperative execution mechanism that can guide the efficient test case generation by synchronizing the symbolic executions of FDMs and concrete executions of device implementations.
3) We propose a novel conformance checking method based on mutant killing and symbolic execution, which enables effective identification of bugs/inconsistencies in implementations and quick measurement of testing adequacy.

Experimental results show that our approach can not only effectively uncover bugs from both virtual devices excerpted from the open source machine virtual platform QEMU [7] and corresponded silicon devices widely used in industry, but also can achieve better test coverage than state-of-the-art methods with less testing efforts.

The rest of this paper is organized as follows. Section 2 introduces background on mutation testing, symbolic execution and conformance checking for computer system designs. Section 3 details our mutation-driven conformance checking method. Section 4 presents experimental results on two industrial network adapters. Finally, Section 5 concludes this paper.

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 adopts 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 virtual prototypes and silicon devices, various methods have been investigated [5], [31], [32], [33]. For example, Lei et al. [31] introduced a conformance checking method that can symbolically execute virtual devices with the same driver request sequences to silicon devices. By extending SystemRDL as a golden reference model, Gu et al. [5] presented a specification-driven symbolic execution approach to check whether the virtual/silicon devices exhibit unexpected behaviors that are not modeled in their given specification. Although the above approaches can efficiently identify inconsistencies in device implementations, few of them consider the quality of conformance checking in terms of testing efficiency. To the best of our knowledge, our approach is the first attempt that utilizes mutated specifications to specify the potential erroneous behaviors and automatically generate effective test inputs to validate the conformance of both virtual and 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.

Specification	Implementation

read(a,b);
$\boxed{\text{MutantCheck((a+b)!=(a-b));}}$
b=a+b;
return b;
	read(a,b);
	b=a-b;
	return b;

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.

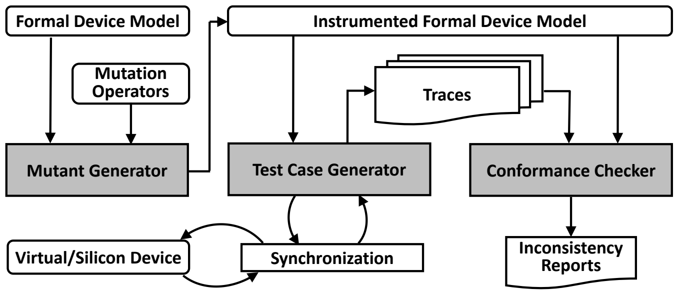


Fig. 2. Workflow of our conformance checking approach

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 pratically 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 "$\rightarrow$", where the left part denotes the correct design in a specification and the right part indicates the potential errors. By default, $RO = \{<, <=, >, >=, ==, != \}$, $LC = \{\&, |, \hat{}\}$, $AO = \{+, -, *, /, \%\}$, and $UO = \{\sim\}$. For example, by using the Arithmetic Operator Replication (AOR) operator we can replace the arithmetic operator α with a new one β, 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 = \{$0xffffffff, 0x00000000, 0x84218421$\}$, and RRAI tries to assign each read register with a specific value from $VAL = \{$0xffffffff$\}$. 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

TABLE 1
Mutant-Killing-Constraint Generation using Our Mutation Operators

Mutation Operators	Mutation Rules	MKC Generation Rules	MKC Generation Examples
ROR (Relational Operator Repl.)	$\forall\alpha.\ \alpha \in RO,\ reg1\ \alpha\ reg2\ \rightarrow$ $\forall\beta.\ (\beta \in RO) \wedge (\beta != \alpha),\ reg1\ \beta\ reg2$	$\forall\alpha.\ \alpha \in RO,\ reg1\ \alpha\ reg2\ \rightarrow$ $\forall\beta.\ (\beta \in RO) \wedge (\beta != \alpha),\ (!(reg1\ \alpha\ reg2) \wedge (reg1\ \beta\ reg2))$ $\vee\ ((reg1\ \alpha\ reg2) \wedge !(reg1\ \beta\ reg2))$	$MutantCheck(\ ((a<b)\ \&\&\ !(a>=b))$ $\Vert\ (!(a<b)\ \&\&\ (a>=b))\ );$ **if(a < b)** ...
LCR (Logical Connector Repl.)	$\forall\alpha.\ \alpha \in LC,\ reg1\ \alpha\ reg2\ \rightarrow$ $\forall\beta.\ (\beta \in LC) \wedge (\beta != \alpha),\ reg1\ \beta\ reg2$	$\forall\alpha.\ \alpha \in LC,\ reg1\ \alpha\ reg2\ \rightarrow$ $\forall\beta.\ (\beta \in LC) \wedge (\beta != \alpha),\ (reg1\ \alpha\ reg2) != (reg1\ \beta\ reg2)$	$MutantCheck(\ (a\&b) != (a\vert b)\ );$ **c = a&b;**
AOR (Arithmetic Operator Repl.)	$\forall\alpha.\ \alpha \in AO,\ reg1\ \alpha\ reg2\ \rightarrow$ $\forall\beta.\ (\beta \in AO) \wedge (\beta != \alpha),\ reg1\ \beta\ reg2$	$\forall\alpha.\ \alpha \in AO,\ reg1\ \alpha\ reg2\ \rightarrow$ $\forall\beta.\ (\beta \in AO) \wedge (\beta != \alpha),\ (reg1\ \alpha\ reg2) != (reg1\ \beta\ reg2)$	$MutantCheck(\ (a+b) != (a-b)\ );$ **c = a + b;**
UOI (Unary Operator Insertion)	$reg\ \rightarrow \forall\mu.\ \mu \in UO,\ \mu\ (reg)$	$reg\ \rightarrow \forall\mu.\ \mu \in UO,\ reg\ != \mu\ (reg)$	$MutantCheck(\ \sim a+1 != \sim(\sim a)+1\ );$ **b =∼ a + 1;**
RRVR (Reserved Register Val. Repl.)	$\forall reg.\ reg \in RSV,\ e \in EXP,\ reg = e$ $\rightarrow\ \forall\upsilon.\ \upsilon \in D,\ reg = \upsilon$	$\forall reg.\ reg \in RSV,\ e \in EXP,\ reg = e\ \rightarrow$ $\forall\upsilon.\ \upsilon \in D,\ (reg == \upsilon) \wedge !(reg == e)$	$MutantCheck(\ r == 0xffffffff$ $\&\&\ r != 0x00000010\ );$ **r = 0x00000010**; //*r* is a reserved reg.
RRAI (Register Read Access Insert.)	$\forall reg.\ reg \in RR,\ return\ reg; \rightarrow$ $\forall\upsilon.\ \upsilon \in VAL,\ return\ \upsilon;$	$\forall reg.\ reg \in RR\ return\ reg; \rightarrow$ $\forall\upsilon.\ \upsilon \in VAL,\ reg != \upsilon;$	$MutantCheck(\ reg != 0xffffffff\ );$ **return reg**; //register read

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 "→" 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 $c = a+b$, 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 "$c = a+b$;" from its mutant "$c = a-b$;".

Based on the rules introduced in Table 1, Listing 1 presents an FDM excerpt of the *e1000* network adapter with instrumented MKCs. To facilitate the illustration, the names of some registers and macros are modified here. In this example, there are three MKCs (lines 5, 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.

```
1  uint32_t runInterfaceFunction(DeviceState* dev, uint32_t
        accType, uint32_t val, uint64_t addr){
2    if(accType == FDM_REG_WRITE){ //register write operation
3      switch(addr){
4        case TCTL_ADDR:
5          MutantCheck((val&0x3ffffff)!=(val|0x3ffffff),1);
6          dev->rega.val=val&0x3ffffff;
7          accFlag=TCTL_ADDR; break;
8        case IMC_ADDR:
9          dev->regb.val=val; accFlag=IMC_ADDR; break;
10       ...}
11   }else{...} // register read operation
12 }
13 void runDevice(DeviceState* dev){
14   switch(accFlag){
15     case TCTL_ADDR:
16       MutantCheck((dev->a&TCTL_EN)!=(dev->a|TCTL_EN),2);
17       if((dev->rega.val&TCTL_EN)==0)
18         return;
19       ...
20     case IMC_ADDR:
21       MutantCheck(~(dev->regb.val))!=~(~(dev->regb.val)),3);
22       uint32_t tmp=~(dev->regb.val);
23       ...
24   }
25 }
```

Listing 1. An FDM excerpt with instrumented MKCs

3.2 Mutant-Driven Test Case Generation

3.2.1 Notations

Interface registers act as the inputs and outputs of devices exposed by device designers. They can be used to feed driver requests and monitor the states of devices. Our test case generation method is based on FDMs which focus on modeling the interface register access behaviors of devices. Note that the usages of interface registers are typically defined at the early stage of the design, i.e., the access behaviors of interface registers in FDMs should be consistently conformed in both virtual and silicon devices.

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 τ 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 τ 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 τ 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 τ 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 τ is a sub-sequence of τ, where $|ts_i| \leq |\tau|$. A continuous test segment sequence of τ in the form of $ts_0 \rightarrow ts_1 \rightarrow \ldots \rightarrow ts_k$ $(k \leq n)$ has the same device request sequence as τ 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 τ with a length of 2, respectively. τ 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 τ is a sequence $\rho = s_0 \xrightarrow{(req_0,\ \mu_0)} s_1 \xrightarrow{(req_1,\ \mu_1)} \ldots s_n \xrightarrow{(req_n,\ \mu_n)} s_{n+1}$, where s_{i+1} denotes the device state after executing req_i from s_i and μ_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 $ts_i = req_{i0} \rightarrow req_{i1} \rightarrow \ldots \rightarrow req_{ik}$ such that we can get $m \in \bigcup_{j=0}^{k} \mu_{ij}$, where $s_i \xrightarrow{(req_{i0},\ \mu_{i0})} s_{i+1} \xrightarrow{(req_{i1},\ \mu_{i1})} \ldots s_{i+k} \xrightarrow{(req_{ik},\ \mu_{ik})} s_{i+k+1}$ 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 $mc = < m_{i,0}, m_{i,1}, \ldots, m_{i,k} >$ $(m_{i,j} \in IM$, $k \geq 0$, $0 \leq j \leq k)$ be a combination of $k+1$ MKCs. A test case τ can cover mc if there exists an initial FDM 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_j can trigger $m_{i,j}$ within the trace of f based on s_0 and τ. We use Mx to denote the test cases generated to cover all the combinations of x MKCs in IM. ■

3.2.2 Synchronization-based Test Case Generation

Since FDMs focus on the modeling of access behaviors of partial interface registers on an abstract level, it cannot be executed like virtual or silicon devices. Therefore, it is impossible to directly generate test cases based on FDMs. According to Definition 3.4, to enable test case generation for Mx we need to figure out the concrete device state for each device request and the continuous test segments to trigger the given combination of MKCs.

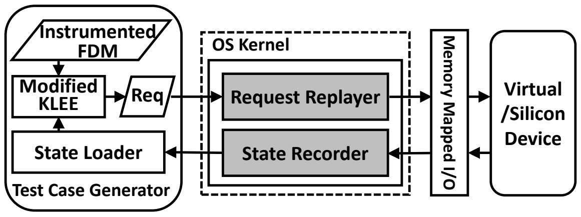


Fig. 3. Test case generation based on FDM-device synchronization

Figure 3 presents our FDM-device synchronization mechanism that enables the test case generation for Mx. In our approach, the test case generator runs on the guest operating systems (OS) of a VM communicating with the underlying virtual/silicon device via our *request replayer* and *state recorder* modules. In Linux, such two modules can be implemented based on the Linux Kernel analysis framework *Kprobes* which provides a means of communication between kernel space and user space. By using symbolic execution engine based on modified KLEE, our test case generator can automate the test case generation for Mx by periodically conducting the following two steps in a synchronous manner: i) use *state loader* to update the abstract device state of the instrumented FDM with the concrete state of corresponding virtual/silicon device; and then ii) generate one device request req using symbolic execution trying to trigger one MKC mu and use *request replayer* to feed it to the virtual/silicon device. Within each synchronization, we use *state recorder* to save interface register values of the virtual/silicon device before and after the device request execution. Meanwhile, *state recorder* also saves the information of both req and mu. Our approach assumes 100 milliseconds for the delay of virtual/silicon device executions. Note that our FDM-device synchronization mechanism enables the test generation from FDM and test execution on virtual/silicon device at the same time. Since our approach operates on interface registers directly based on the memory mapped I/O mechanism, to avoid the interference from other network applications, we unload all the correlated device drivers from the guest OS.

3.2.3 FDM Harness for Test Case Generation

Our approach adopts FDMs as our device specifications, which provide a template to model the access behaviors of interface registers [5]. To enable the symbolic execution based test case generation, an FDM should have a harness. Listing 2 shows the harness of an instrumented FDM f for the generation of device requests to trigger MKCs. Note that all the device FDMs have the same harness. After updating interface registers of f in line 3, lines 4-12 iteratively search for device requests for MKCs not yet covered. According to Definition 3.1, the harness defines three variables acc, $addr$ and val indicating the three key elements of a device request, which are made symbolic using the function *klee_make_symbolic* for device request generation.

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 while-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.

```
1  int main(int argc, char *argv[]){
2    struct DevState dev; int cnt=0;
3    getDevState(&dev);
4    while(cnt<2){
5      uint32_t acc,val; uint64_t addr; cnt++;
6      klee_make_symbolic(&acc,sizeof(acc),"accessType");
7      klee_make_symbolic(&addr,sizeof(addr),"regAddr");
8      klee_make_symbolic(&val,sizeof(val),"value");
9      runInterfaceFunction(&dev, acc, val, addr);
10     runDevice(&dev);
11     terminateState();}
12     ...
13 }
```

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_x to indicate the x^{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.

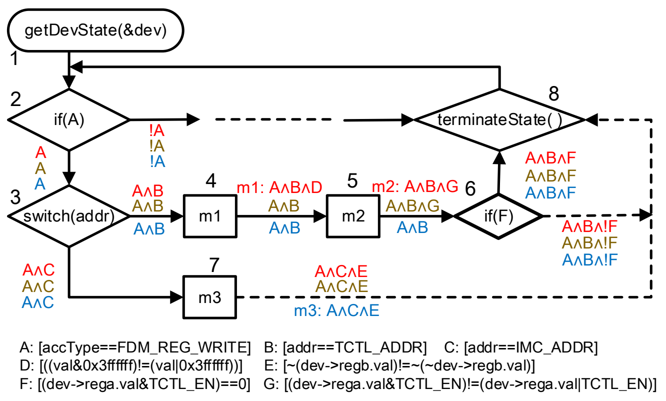


Fig. 4. An illustration of $M1$ test case generation for FDM in Listing 1

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 three 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 $mutComCov$=$\{<m1>,<m2>,<m3>\}$ 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) ρ forks two execution paths (i.e., $\rho1 = 1 \rightarrow 2 \rightarrow \ldots,\ \rho = 1 \rightarrow 2 \rightarrow 3$). Since we omit the part between node 2 and node 8, in this example $\rho1$ will not be discussed. Assuming that ρ has a high priority, $\rho1$ will be saved temporarily. When our symbolic execution hits the node 3, ρ will be forked again into two execution paths (i.e., $\rho = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4,\ \rho2 = 1 \rightarrow 2 \rightarrow 3 \rightarrow 7$). Assuming that ρ has a higher priority, $\rho2$ will also be saved for future analysis. When the search of ρ 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 $tList = \{m1 : A \wedge B \wedge D\}$ assuming that $A \wedge B \wedge D$ is satisfiable. After that, ρ will hit m_2 at node 5. By checking $mutComCov$, we will update $tList = \{m1 : A \wedge B \wedge D, m2 : A \wedge B \wedge G\}$ assuming that $A \wedge B \wedge G$ is satisfiable. When ρ hits node 6, it will fork two execution paths (i.e., $\rho = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 8,\ \rho3 = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow \ldots$). Assuming that ρ has a high priority, $\rho3$ will not be discussed due to lack of successive nodes. Thereafter, ρ will hit node 8, where it will be suspended temporarily. After that $\rho2$ will be explored. When $\rho2$ hits node 7, after checking $mutComCov$, $tList$ will be updated to $\{m1 : A \wedge B \wedge D, m2 : A \wedge B \wedge G, m3 : A \wedge C \wedge E\}$. When all the active execution paths finish the exploration, the *terminateState()* will terminate the first symbolic execution and select $m1$ 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 $\{<m2>,<m3>\}$. Similarly, the second symbolic execution explores the harness function in the same way. The only difference is that $m1 : A \wedge B \wedge D$ will not be added to $tList$, since $<m1>$ is not in $mutComCov$. In this case, a device request for $m2$ will be generated. In the last round, $m3$ 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 $rndRunCnt$ (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      ResetDevice(dev);
4      rndRunCnt=0, reqCnt=0;
5      reqEventSeq.Clear();
6      while rndRunCnt < rndBound do
7          curSta = GetRegState(dev);
8          executionState = ExploreState(fdm, curSta, mutSet, k);
9          if executionState != NULL then
10             muID = executionState.idMKC;
11             req = DeviceRequestGen(executionState);
12             SendRequestToDevice(req);
13             nxtSta = GetRegState(dev);
14             killedMutSeq.Append(muID);
15             if reqCnt == k then
16                 reqCnt=0;
17                 mutComCov.Add(killedMutSeq);
18                 TCS.Add(reqEventSeq);
19                 reqEventSeq.Clear();
20                 killedMutSeq.Clear();
21             end
22             else
23                 reqCnt++;
24                 reqEventSeq.Append(<
                   req, curSta, nxtSta, muID >);
25             end
26             rndRunCnt = 0;
27         end
28         else
29             RandomRun(dev);
30             rndRunCnt += 1;
31         end
32         if rndRunCnt % rstBound == 0 then
33             ResetDevice(dev);
34         end
35     end
36     return TCS;
37 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 $< req, curState, nxtSta, muID >$ indicating the device request, current device state, next device state, and the ID of triggered mutant by *req*, respectively. We call such a record an event of the device request. Within an FDM-device synchronization, Line 7 initializes the interface registers using the device state. Line 8 resorts to $ExploreState()$ function (see Algorithm 2) to conduct the symbolic execution. It will obtain a most suitable execution state that can be used to generate a device request for a yet-to-cover MKC. If such an execution state exists, lines 10-14 will generate the device request and feed it to the device to obtain the new device state for next device request generation. Since an M_k test case is a sequence of device requests, line 14 updates a global variable *killedMutSeq* to indicate such a sequence searched so far. If the sequence saved in *killedMutSeq* can trigger a combination of k MKCs, lines 15-21 will update the k-MKC combination coverage information in *mutComCov*, save the generated test case, and clear the data structures for a new test case generation. Finally, the algorithm returns all the generated M_k test cases.

Algorithm 2: Algorithm for ExecutionState Selection

```
   Input: i) fdm, an instrumented FDM ;
          ii) curState, starting state of registers;
          iii) mutSet, set of all MKCs;
          iv) k, # of mutants in a combination of $M_k$;
   Output: selExecState, an execution state triggering a new MKC
1  ExploreState(fdm, curState, mutSet, k) begin
2      availMKC = mutSet, socre[mutSet.Size() + 1] = 0;
3      selExecState = NULL, tList.Clear();
4      if killedMutSeq.Size() == k − 1 then
5          visited.Clear();
6          for i from 0 to mutComCov.Size()-1 do
7              if killedMutSeq.Equals(mutComCov[i].SubSeq(k − 1))
               then
8                  visited.Add(mutComCov[i][k − 1]);
9              end
10         end
11         availMKC = availMKC - visited;
12     end
13     ES = SymbolicExecute(fdm, curState);
14     for i from 0 to ES.Size()-1 do
15         es = ES[i];
16         MKC = es.TraversedMKC();
17         for j from 0 to MKC.Size()-1 do
18             if MKC[j] ∈ availMKC then
19                 if SAT(es.pathConstr ∧ MKC[j].constr) then
20                     es.idMKC = MKC[j].ID;
21                     availMKC -= MKC[j];
22                     tList += es; break;
23                 end
24             end
25         end
26     end
27     for i from 0 to mutComCov.Size()-1 do
28         for j from 1 to k do
29             score[mutComCov[i][j]]++;
30         end
31     end
32     if !tList.IsEmpty() then
33         selExecState = Sort(tList, score);
34     end
35     return selExecState;
36 end
```

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*, lines 6-10 will ignore the MKC. Note that lines 6-10 iteratively compare *killMutSeq* with the explored MKC combinations in *mutComCov*. At line 7, if *killMutSeq* with a length of $k-1$ is a prefix of *mutComCov*$[i]$, line 8 will record the last MKC of *mutComCov*$[i]$ in *visited*. Line 11 figures out unexplored MKCs which can lead to new k-MKC combinations, and save them in *availMKC*. After the full symbolic execution on the given FDM with a start state, line 13 collects all the execution states and saves them in *ES*. Similar to the *tList* construction process illustrated in Figure 4, lines 14-26 construct a candidate execution state list based on uncovered MKCs. Line 16 collects all the MKCs that are satisfiable at current 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 R_I and R_N 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 execution of 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 $CS =< ID_\mu, \{S_I, S_N\} >$ where ID_μ 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 $CF_p =< IDS_\mu, \{F_I, F_N\} >$, where IDS_μ 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 $Symb(S)$ and $Symb(F)$ 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 $CS =< ID_\mu, \{S_I, S_N\} >$ and a checked state of corresponding FDM $CF_p =< IDS_\mu, \{F_I, F_N\} >$ are consistent to each other if both of $Symb(\{S_I, S_N\}) \cap Symb(\{F_I, F_N\}) \neq \emptyset$ and $ID_\mu \in IDS_\mu$ 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 *choice*(), our approach can reflect the real hardware behaviors by executing functions *runInterfaceFunction()* and *runDevice()* (see details in Listing 1) in a non-deterministic manner.

```
int main(int argc, char *argv[]){
  struct DevState dev;
  uint32_t acc,val;
  uint64_t addr;
  while(choice()){
   switch(choice()){
    case 0 :
      runInterfaceFunction(&dev, acc, val, addr);
      break;
    case 1:
      runDevice(&dev);
      break;
   ...}
  }
 ...
}
static inline int choice(){
  ...
  make_symbolic(&i, sizeof(i), "choice");
  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 $< muID, nxtSta >$ 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          $< req, curSta, nxtSta, muID >$ = 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.

TABLE 2
Settings of Intel Network Adapters

Devices	FDM (LoC)	iFDM (LoC)	# of MKCs	VP (LoC)	Selective Captured Size (Bytes)
e1000	473	588	115	1731	1224
eepro100	525	646	121	2115	74

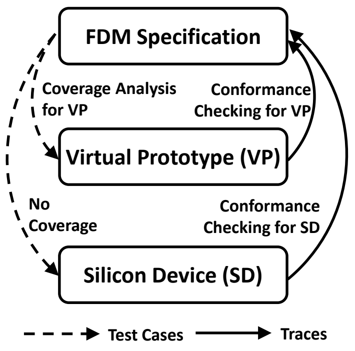


Fig. 5. Conformance checking relations of three different design layers

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 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

Mutation Methods	*e1000* FDM			*eepro100* FDM		
	Total	Succ.	Fail.	Total	Succ.	Fail.
M1	115	112	3	121	83	38
M2	13225	12700	525	14641	7074	7567

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 Mx 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 *eepro100* 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
Time and Coverage Information of Virtual Devices Testing

Devices	Test Cases	Time	Line Cov. (%)	Func. Cov. (%)	Branch Cov. (%)
e1000	Com [5]	25m	79.9	81.4	55.8
	M1	5m	75.5	90.7	47.5
	M2	6h53m	88.0	95.3	66.0
	M1+M2	6h57m	88.0	95.3	66.0
	M1+M2+Com	7h20m	88.0	95.3	66.0
eepro100	Com [5]	15m	68.8	73.8	42.2
	M1	1m20s	71.2	73.8	47.3
	M2	1h20m	71.2	73.8	47.7
	M1+M2	1h22m	71.2	73.8	47.7
	M1+M2+Com	1h40m	71.2	73.8	47.7

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.

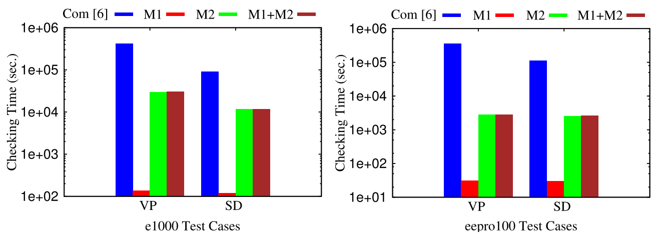
From Table 4, we can observe that $M1$ spends much less testing time than *Com*. This is because *Com* executes more than one million device requests for each design, while $M1$ only generates and executes 112 device requests for *e1000* and 83 device requests for *eepro100*, though the majority of testing efforts involving $M1$ and $M2$ are spent on test case generation. However, from *eepro100* 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 *eepro100 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 *eepro100* 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.



(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

Mutation Methods	**Mutant Coverage / Line Coverage (%)**			
	e1000-VP	**e1000-SD**	**eepro100-VP**	**eepro100-SD**
Com [5]	39.13/63.40	41.74/58.49	64.46/76.59	63.64/83.33
M1	96.52/97.74	93.04/90.94	66.94/84.92	67.77/84.92
M2	96.52/98.49	94.78/92.83	66.94/84.52	67.77/84.92
M1+M2	96.52/98.49	94.78/92.83	66.94/84.92	67.77/84.92

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,

TABLE 6
Inconsistencies Identified from Virtual/Silicon Devices

Type	**Description**	***e1000* Bug #**		***eepro100* Bug #**	
		SD	VP	SD	VP
E1	Update reserved SD registers	**3/3/3**	0/0/0	0/0/0	0/0/0
E2	Update reserved VP registers	0/0/0	**1/1/1**	0/0/0	0/0/0
E3	Generate unnecessary interrupts	0/0/0	**1/1/1**	0/0/0	0/0/0
E4	Fail to update necessary registers	0/0/0	**1/1/1**	0/0/0	**1/1/1**
E5	Write incorrect values to registers	0/0/0	**3/3/3**	0/0/0	0/0/0

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 *0x1420000*. Our framework can detect the inconsistency where the register ICR at address *0xC0* in virtual device has a value of *0x80000200*, while the ICR register in the FDM specification has a value of *0x0*. According to the specification, if the 29th bit[1] 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 use 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 7
An Example of Register Value Inconsistency

Request	Write MDIC (addr=0x20) with value: 0x14200000		
Inconsistent Info.		Value in Virtual Device	Value in Specification
	ICR (addr=0xC0)	0x80000200	0x00000000

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

1. The lowest bit of the address is the 0th bit.

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

[1] O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. Mueller-Gritschneder, P. Sasidharan and S. Singh, "The next generation of virtual prototyping: ultra-fast yet accurate simulation of HW/SW systems," in *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015, pp. 1698–1707.

[2] M. M. Swift, M. Annamalai, B. N. Bershad and H. M. Levy, "Recovering device drivers," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004, pp. 1–16.

[3] P. Mishra, R. Morad, A. Ziv and S. Ray, "Post-silicon validation in the SoC era: A tutorial introduction," *IET Computers & Digital Techniques*, vol. 34, no. 3, pp. 68–92, 2017.

[4] SystemRDL 1.0 Standard, http://www.accellera.org/downloads/standards /systemrdl.

[5] H. Gu, M. Chen, T. Wei, L. Lei and F. Xie, "Specification-driven automated conformance checking for virtual prototype and post-silicon designs," in *Proceedings of Design Automation Conference (DAC)*, 2018, pp. 93:1–93:6.

[6] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. Halleux and H. Mei, "Test generation via Dynamic Symbolic Execution for mutation testing", in *Proceedings of International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.

[7] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2005, pp. 41–46.

[8] M. Chen, X. Qin, H. Koo and P. Mishra, "System-Level Validation: High-Level Modeling and Directed Test Generation Techniques," *Springer*, 2013.

[9] J. Li, F. Xie, T. Ball, V. Levin, C. McGarvey, "Formalizing hardware/software interface specifications," in *Proceedings of International Conference on Automated Software Engineering (ASE)*, 2011, pp. 143–152.

[10] N. Bombieri, F. Fummi, G. Pravadelli and J. Marques-Silva, "Towards equivalence checking between TLM and RTL models," in *Proceedings of International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, 2007, pp. 113–122.

[11] M. Chen and P. Mishra, "Assertion-based functional consistency checking between TLM and RTL models," in *Proceedings of International Conference on VLSI Design*, 2013, pp. 320–325.

[12] P. Herber, M. Pockrandt, and S. Glesner, "Automated conformance evaluation of SystemC designs using timed automata," in *Proceedings of IEEE European Test Symposium (ETS)*, 2010, pp. 188–193.

[13] P. Godefroid, M. Y. Levin and D. A. Molnar, "SAGE: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[14] V. Chipounov, V. Kuznetsov and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 265–278.

[15] C. Cadar, D. Dunbar and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008, pp. 209–224.

[16] K. Sen, D. Marinov and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2005, pp. 263–272.

[17] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara and F. Xie, "CRETE: A versatile binary-level concolic testing framework," in *Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2018, pp. 281–298.

[18] S. Guo, M. Wu, and C. Wang, "Symbolic execution of programmable logic controller code," in *Proceedings of Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 326–336.

[19] A. Offutt, G. Rothermel and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of International Conference on Software Engineering (ICSE)*, 1993, pp. 100–107.

[20] M. Papadakis, D. Shin, S. Yoo and D. Bae, "Are mutation scores correlated with real fault detection? A large scale empirical study on the relationship between mutants and real faults," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2018, pp. 537–548.

[21] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon and M. Harman, "Mutation Testing Advances: An Analysis and Survey," *Advances in Computers*, vol. 112, pp. 275–378, 2019.

[22] M. Papadakis and N. Malevris, "Automatic Mutation Test Case Generation via Dynamic Symbolic Execution," in *Proceedings of International Symposium on Software Reliability Engineering (ISSRE)*, 2010, pp. 121–130.

[23] M. Papadakis and N. Malevris, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Software Quality Journal*, vol. 19 no. 4, pp. 691–723, 2011.

[24] M. J. Renzelmann, A. Kadav and M. M. Swift, "SymDrive: Testing drivers without devices," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 279–292.

[25] T. Yu, "An observable and controllable testing framework for modern systems," in *Proceedings of International Conference on Software Maintenance (ICSE)*, 2013, pp. 1377–1380.

[26] L. G. Murillo, R. L. Buecs, R. Leupers, and G. Ascheid, "MPSoC Software Debugging on Virtual Platforms via Execution Control with Event Graphs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 1, 7, 2016.

[27] T. Yu, W. Srisa-an and G. Rothermel, "SimTester: a controllable and observable testing framework for embedded systems," in *Proceedings of ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE)*, 2012, pp. 51–62.

[28] T. Yu, W. Srisa-an and G. Rothermel, "An automated framework to support testing for process-level race conditions," *Software Testing, Verification & Reliability*, vol. 27. no. 4-5, pp. 1-26, 2017.

[29] Y. Wang, L. Wang, T. Yu, J. Zhao and X. Li, "Automatic detection and validation of race conditions in interrupt-driven embedded software," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, 2017, pp. 113–124.

[30] T. Yu, X. Qu and M. B. Cohen, "VDTest: an automated framework to support testing for virtual devices," in *Proceedings of International Conference on Software Maintenance (ICSM)*, 2016, pp. 583–594.

[31] L. Lei, F. Xie, and K. Cong, "Post-silicon conformance checking with virtual prototypes," in *Proceedings of Design Automation Conference (DAC)*, 2013, pp. 29:1–29:6.

[32] K. Cong, F. Xie and L. Lei, "Symbolic Execution of Virtual Devices," in *Proceedings of International Conference on Quality Software (QSIC)*, 2013, pp. 1–10.

[33] K. Cong, F. Xie and L. Lei, "Automatic concolic test generation with virtual prototypes for post-silicon validation," in *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, 2013, pp.303–310.

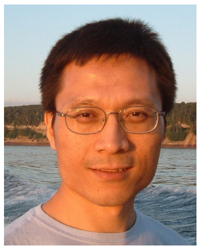
[34] A. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, 118, 1996.

[35] R. DeMillo and A. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, 1991.
[36] W. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE Transactions on Software Engineering*, vol. 8, no. 4, pp. 371–379, 1982.
[37] "Milu", https://github.com/yuejia/Milu.
[38] "PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual", https://www.intel.com/content/dam/doc/manual/pci-pci-x-family-gbe-controllers-software-dev-manual.pdf.
[39] "Intel 8255x 10/100 Mbps Ethernet Controller Family Software Developer Manual", https://www.intel.com/content/dam/doc/manual/8255x-10-100-mbps-ethernet-controller-software-dev-manual.pdf.
[40] "e1000: eliminate infinite loops on out-of-bounds transfer start," https://lists.gnu.org/archive/html/qemu-devel/2016-01/msg03454.html.
[41] "e1000: Avoid infinite loop in processing transmit descriptor (CVE-2015-6815)," https://lists.gnu.org/archive/html/qemu-devel/2015-09/msg03983.html.
[42] R. H. Untch, A. J. Offutt and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, 1993, pp. 139–148
[43] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen and A. Groce, "Mutation Reduction Strategies Considered Harmful," *IEEE Transactions on Reliability*, vol. 66, no. 3, pp. 854–874, 2017.
[44] P. Ammann, M. E. Delamaro and J. Offutt, "Establishing Theoretical Minimal Sets of Mutants," in *IEEE International Conference on Software Testing, Verification, and Validation (ICST)*, 2014, pp. 21–30.
[45] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E.W. Krauser, R.J. Martin, A. P. Mathur, E. Spafford, "Design of Mutant Operators for the C Programming Language," *Techreport, Purdue University*, 1989.

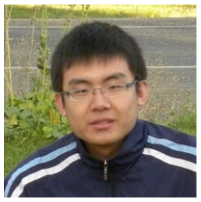
Haifeng Gu received the BE degree from the Department of Computer Science and Technology, Sichuan Normal University, Chengdu, China, in 2013. He is currently working toward the PhD degree in the Department of Embedded Software and System, East China Normal University, Shanghai, China. His research interests 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.

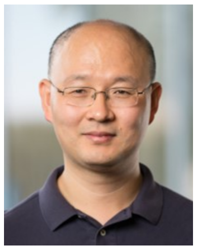
Mingsong Chen (M'08–SM'17) received the B.S. and M.E. degrees from Department of Computer Science and Technology, Nanjing University, Nanjing, China, in 2003 and 2006 respectively, and the Ph.D. degree in Computer Engineering from the University of Florida, Gainesville, in 2010. He is currently a Professor 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.



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.



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.