Specification-Driven Automated Conformance Checking for Virtual Prototype and Post-Silicon Designs

Haifeng Gu[1], Mingsong Chen[1], Tongquan Wei[1], Li Lei[2], Fei Xie[3]
[1]Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai 200062, China
[2]Intel Labs, Hillsboro, OR 97124, United States
[3]Department of Computer Science, Portland State University, Portland, OR 97207, United States

ABSTRACT

Due to the increasing complexity of System-on-Chip (SoC) design, how to ensure that silicon implementations conform to their high-level specifications is becoming a major challenge. To address this problem, we propose a novel specification-driven conformance checking approach that can automatically identify inconsistencies between different levels of designs. By extending SystemRDL specifications, our approach enables the generation of high-level Formal Device Models (FDMs) that specify access behaviors of interface registers triggered by driver requests. Based on the symbolic execution of the generated FDMs with the same driver requests to virtual/silicon devices, our approach can efficiently check whether the designs of an SoC at different levels exhibit unexpected behaviors that are not modeled in the given specification. Experiments on two industrial network adapters demonstrate the effectiveness of our approach in troubleshooting bugs caused by inconsistencies in both virtual and post-silicon prototypes.

1 INTRODUCTION

System-on-Chips (SoCs) are increasingly designed at higher level of abstraction in order to handle their growing complexity. Typically, a top-down SoC design flow starts from high-level specifications. After design space exploration based on these specifications, the optimized design goes through virtual prototyping. Since virtual prototyping enables the co-development of system hardware/software components in parallel, it has been increasingly used in driver development when host devices are not ready [1]. As a notable case, Intel developed its 40 Gigabit Ethernet adapter on top of virtual devices before the FPGA prototype is available [2].

The goal of virtual prototyping is to reduce the product time-to-market. However, to achieve this benefit, there are two key challenges that need to be addressed. The first one is the lack of formal specifications that can be consistently used across the different stages of product lifecycle (from design, to pre-silicon, to post-silicon). Since virtual prototyping involves lots of human efforts, the implemented virtual devices are inevitably error-prone. If there is no consistent view for the interactions between hardware and software components, it is hard to ensure the correctness of virtual prototypes. The second one is the lack of effective conformance checking tools to identify inconsistent implementations in virtual and silicon devices. As silicon devices do not always conform to their virtual counterparts, drivers developed for virtual prototypes often cannot readily work on silicon devices, causing serious problems (e.g., system crashes [3]). Especially when the controllability and observability of silicon designs are limited, the errors caused by inconsistent implementations cannot be easily detected by existing ad-hoc validation approaches at the post-silicon stage [4].

As a semi-formal register description language, SystemRDL [5] has become the de-facto standard for describing and managing registers in SoC design. Besides register metadata modeling for high-level designs, SystemRDL defines registers in downstream hardware/software implementations. Therefore, SystemRDL provides a systematic and consistent view into registers along the whole SoC design flow. Although SystemRDL can be used to conduct conformance checking, its current version only supports the structural definition of registers. In other words, it only enables the conformance checking of syntactical usages of registers rather than system behaviors exhibited at different design levels.

To enable bug-free SoC implementations, we propose a novel approach that can automatically conduct conformance checking between high-level specifications (i.e., SystemRDL) and low-level implementations (i.e., virtual and silicon devices). This work makes the **following three major contributions**: i) We invent novel extensions which significantly increase the expressive capacity of SystemRDL. ii) We propose an automatic approach for bridging the gaps between high-level specifications to golden reference models. iii) Based on the symbolic execution, we present a comprehensive framework for validating implementation prototypes across different phases of product lifecycle. Experimental results show that our approach can effectively identify real bugs from both virtual devices excerpted from the QEMU virtual machine [6] and corresponding industrial silicon devices, some of which cannot be discovered by state-of-the-art approaches.

The rest of this paper is organized as follows. Section 2 introduces the relevant background. Section 3 proposes the details of our conformance checking approach. Section 4 reports the performance evaluation results. Finally, Section 5 concludes this paper.

2 RELATED WORKS

Specification-driven approaches have been increasingly used in conformance checking between different abstraction layers of SoC design. For example, an event-based approach that supports the equivalence checking between Transaction Level Modeling (TLM) and Register Transfer Level (RTL) designs is presented in [8]. In [9], an effective Property Specification Language (PSL)-based assertion refinement approach is proposed to enable the consistency checking between TLM and RTL designs. Based on timed automata, a conformance test generation method to check the consistency between highly abstract models and detailed implementations in SystemC is introduced in [10]. However, few of existing approaches enable the consistency checking for both virtual and silicon devices.

Symbolic execution has been successfully used in testing software and hardware components. For example, the tools SAGE [11] and S2E [12] adopted symbolic execution to test software systems, which

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196119

intensively interact with external environments. Based on symbolic execution and constraint solvers, the tools KLEE [13], CUTE [14] and the one developed in [15] can generate high-quality test cases for hardware/software designs. However, most of these techniques focus on testing rather than conformance checking.

Our approach is inspired by the conformance checking method presented in [7], which symbolically executes virtual devices with the same driver request sequences to silicon devices. Although the approach can identify inconsistencies between virtual and silicon devices, due to the lack of formal models, the correctness of silicon devices cannot be clearly judged. As far as we know, our approach is the first attempt to establish a golden reference model which can effectively guide the correctness and consistency checking for both virtual and silicon devices.

3 CONFORMANCE CHECKING

Figure 1 presents the workflow of our specification-driven conformance checking approach. It has three major components: a trace recorder, a Formal Device Model (FDM) generator and a conformance checker. By registering our developed callback functions to the kernel API interceptor, our trace recorder can be used to capture both the driver requests and device states from silicon devices or virtual prototypes. We implemented the FDM generator using the tool ANTLR, which can parse our extended SystemRDL specifications and transform them into executable FDM models. Our conformance checker is implemented on top of a symbolic virtual machine named KLEE [13]. It can replay a sequence of driver requests on the derived executable FDMs to enable the conformance checking. The following subsections will introduce our approach in detail.

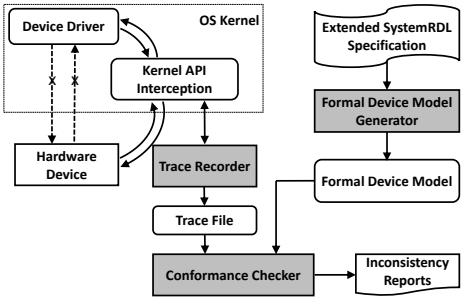


Figure 1: Workflow of our conformance checking method

3.1 Syntax Extensions for SystemRDL

The original version of SystemRDL is designed to describe the structural information of various registers. To model the dynamic behaviors of register accesses, we propose both syntax and semantics extensions for SystemRDL specifications. Since our extended SystemRDL specification has a consistent view of register usages across different design layers, it is very suitable for conformance checking.

To present the syntax extension of SystemRDL, we explain all the notations of extensions using the *Extended Backus-Naur Form*, where keywords and terminal symbols are printed in bold; alternatives are separated by "|"; groupings are enclosed with parentheses "( )"; square braces "[ ]" delimit *optional elements*; and "{ }+" and "{ }∗" are used to denote *one-or-more*, and *zero-or-more* of the enclosed elements, respectively. The following rules show the component skeleton for the extended SystemRDL. To enable the global replacement of variables with specific values, we adopt the *macro* construct that is similar to the macro in the C++ programming language. To allow designers to specify the access behaviors of interface registers in SystemRDL, we introduce the *function* construct, which has the similar syntax as the functions in C++. In a SystemRDL function, we can update the value of an interface register by using a C++ like assignment statement.

component_def ::= *new_comp_body* | *component_type*
component_name **{**{[*component_def***;**]* [*property***;**][...**;**]}∗**};**
component_type ::= **field** | **reg** | **regfile** | **addrmap**
| **signal** | **enum** | **macro** | **function**
property ::= *property_name* **=** *value***;**
new_comp_body ::= *comp_body* | *macro_body* | *function_body*

We incorporate these two new notations as two new components in SystemRDL by adding the key words **macro** and **function** in rule *component*, and defining their syntax details in the following two new rules i.e., *macro_body* and *function_body*. As an extension, the components of type **macro** and **function** can be embedded into original SystemRDL specifications to specify the operations on registers (see Figure 2(a) for more details). Due to space limitation, we do not present the details of the rule for *statements*, which has the same definition as the C++ programming language.

macro_body ::= {*macro_name* **=** *value***;**}∗
function_body ::= *fun_type fun_name* **(** *argument_list* **) {**
statements **}**

3.2 Automated Generation of FDMs

Although extended SystemRDL can be used as a semi-formal specification to model both the structural relations and access behaviors of interface registers, it cannot be directly used for conformance checking since the specification itself is not executable. In this subsection, we present a set of transformation rules that can translate extended SystemRDL specifications to executable FDMs automatically. As an executable model, an FDM mainly consists of two parts: a harness module that acts like the main function of an FDM, and a register operation module that consists of all the necessary operations on interface registers. Note that both the harness module and register operation module can be automatically generated by our FDM generator. This subsection only presents the transformation from extended SystemRDL designs to FDMs. The FDM execution details will be presented in Section 3.4.

3.2.1 Generation of Harness Module. Unlike software which is executed sequentially, the hardware components of a system are executed in parallel. To reflect the real hardware behaviors using software, we resort to symbolic execution that can model non-deterministic software behaviors [13] by enumerating all the possible execution interleavings. The harness module plays an important role in symbolically executing FDMs. Acting like a scheduler, the harness module abstracts the choice of operations on interface registers, which enables the interleaved executions of register operations. Note that our approach separates the non-deterministic execution process and the concrete register update operations into two disjoint parts. In other words, different FDMs have the same harness modules.

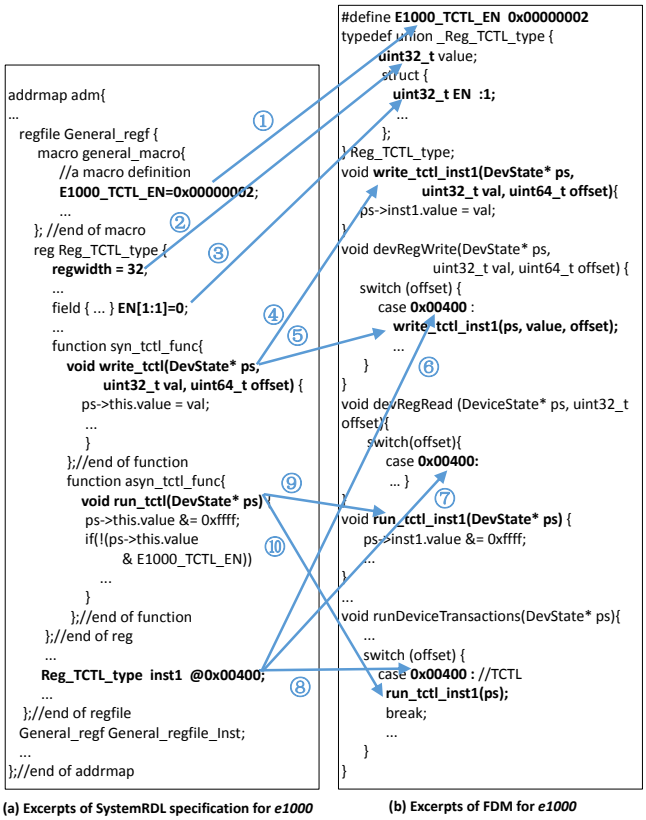
Due to the space limitation, Listing 1 only presents a skeleton of our harness module. Note that to enable the non-deterministic execution, we create a queue data structure in the harness module, which can temporarily hold the incoming driver requests without executing it. The main body of the harness module is a while loop. Within a loop iteration, one of the following three kinds of operations will be invoked: i) the transaction *runInterfaceFunction* that reads/writes the

given registers as in driver requests; ii) the asynchronous transaction *runDevice* that responds to the driver request at the queue head; iii) a void operation doing nothing. To enumerate all possible interleavings of both synchronous and asynchronous transactions, we use the symbolic value returned by the inline function *fdm_choice* to guide the symbolic execution of the while loop.

```
1  ...
2  while(fdm_choice()){
3   switch(fdm_choice()){
4      case 0 : //synchronous transaction
5          runInterfaceFunction(fdm_State, deviceEntry,
6                                  val, offset_addr);
7          break;
8      case 1: //asynchronous transaction
9          runDevice(fdm_State);
10         break;
11     default: //Do nothing
12         break;
13    }
14 }
15 ...
16 static inline int fdm_choice(){
17     ...
18     fdm_make_symbolic(&i,sizeof(i), "choice");
19     return i;
20 }
21 ...
```

Listing 1: Excerpts of our FDM harness module

3.2.2 Generation of Register Operation Module. While Section 3.2.1 presents a non-deterministic operation invocation framework for interface registers, this subsection introduces the construction process of a register operation module that comprises of a set of register operations generated from a given SystemRDL specification. Note that different FDMs have different register operation modules.



(a) Excerpts of SystemRDL specification for *e1000* (b) Excerpts of FDM for *e1000*

Figure 2: Transformation from extended SystemRDL to FDM

By parsing the syntax of an extended SystemRDL specification, our FDM generator can figure out its component organization. Based on the meaning of each component, our tool can automatically generate the register operation module. Figure 2 shows an example that converts the extended SystemRDL specification of Intel *e1000* network adaptor to its corresponding FDM. We use this example to introduce our transformation rules, since it covers majority of the FDM constructs that need to be converted. As illustrated in Figure 2, the register file *regfile* contains two components: i) one macro named *general_macro*; and ii) a register definition named *Reg_TCTL_type* where *reg_TCTL_inst* is an instance of this register type with an offset address 0*x*00400. Note that the register operation module of each FDM specification contains four reserved functions, i.e., *devRegWrite* and *devRegRead* that can only be called by *runInterfaceFunction* function in the harness module, and *runDeviceTransactions* and *runEnvironment* that can only be called by *runDevice* function in the harness module.

As defined in Section 3.1, **macro** is an extended component for systems. It is used to facilitate the programming of the following **function** components. Within a macro component, we can define multiple macro assignments. During the transformation, all macro assignments will be converted to the corresponding macros in C++. For example, the mapping 1 in Figure 2 shows an example of the conversion from a SystemRDL macro assignment to a C++ macro.

The register definition *Reg_TCTL_type* in the *e1000* SystemRDL specification provides both the property and operation details for the specific register type, which will be converted into the register operation module. For each register type in the specification, we create one corresponding union type in FDM. The properties of the register type should also be converted accordingly. For example, mapping 2 converts a register with a width of 32 bits. In the union construct, we create a variable named *value* to hold the register value with a size of *uint32_t*. Mapping 3 deals with the filed component of a register. In the mapped FDM implementation, we use the member variable of a struct within the union type to name the specific bits of some register. Mappings 4-5 conduct the transformation of a synchronous function (indicated by *syn_tctl_func*) defined in a nested **function** components. The conversion involves two steps, where mapping 4 is to create one corresponding operation function in FDM with the name *write_tctl_inst1*, and mapping 5 incorporates this function in the function *devRegWrite*. Since the offset address of the register *inst1* is *0x00400*, this information should be reflected in all the reserved functions as shown by mappings 6-8. Mappings 9-10 deal with the transformation of an asynchronous function named *run_tctl* defined in a nested **function** components. Mapping 9 constructs a new function named *run_tctl_inst1* and mapping 10 integrates this function to the reserved function *runDeviceTransactions*. Based on the above 10 mapping rules, we can automatically transform the extended SystemRDL specifications to corresponding executable FDMs.

3.3 Device Trace Collection

In our approach, the conformance checking between FDMs and devices is based on the symbolic execution of device traces on the executable FDMs. Since virtual prototypes and silicon devices can dump their states in terms of interface registers triggered by the drive requests, our approach supports the conformance checking for both design layers. In our approach, all the traces of virtual/silicon devices are dumped by the same hook functions instrumented in operating system kernels. Therefore, the trace format of both virtual and silicon devices are same.

DEFINITION 3.1. *Let R_I and R_N be the set of interface registers and internal registers, respectively. A state of the device is represented as $S = \{S_I, S_N\}$ where S_I indicates the assignments to the set R_I and S_N indicates the assignments to the set R_N.* ■

DEFINITION 3.2. *Let<S_{I_k},A_k> ($0 \leq k \leq n$) be a 2-tuple for device-states and driver requests, where S_{I_k} denotes the current device interface state and A_k denotes the forthcoming driver request. A device trace can be represented by a sequence of such 2-tuples in the form of T = <S_{I_0},A_0>,<S_{I_1},A_1>,...,<S_{I_n},A_n>.* ■

Definition 3.1 presents the formal definition of devices states, which denote the assignments to both interface and internal registers. Definition 3.2 formally presents the structure of device traces. In this definition, a trace consists of a sequence of <*state,action*> pairs, where *state* denotes the current assignments to all the registers (i.e., R_I and R_N) and *action* indicates a register update operation triggered by the forthcoming driver request. For example, when sending a *ping* command to the virtual/silicon device of Intel *e1000* network adapter, the command will be converted into a sequence of driver requests, where each request tries to read or write a specific device interface register.

To manipulate hardware devices, drivers need to invoke operating system kernel functions to access their interface registers. Therefore, the driver request sequence and the device states (i.e., values of interface registers) can be recorded by proper function hooking mechanisms using user-defined callback functions. In our experiment, we collected the traces for both virtual and silicon devices on a Linux machine. By using the Linux kernel analysis framework *Kprobes*, we implemented a trace recorder as a Linux kernel module, which can break into the kernel functions (e.g., *iowrite32*, *ioread32*) revoked by the specified device driver. Our trace recorder can dump the state S_{I_k} of a given device right before issuing the driver request A_k. Since complex SoC designs usually consist of a large number of the interface registers, dumping all their values will be time- and memory-consuming. Meanwhile, the performance of the host machine can be significantly degraded. Therefore, conformance checking methods adopt the selective recording of interface registers. To enable the selective recording, our trace recorder allows the users to dump device traces based on their specified interface register ranges.

3.4 Conformance Checking with FDM

Since FDMs are executable models that are automatically generated from the semi-formal SystemRDL specifications, they can be used as the golden reference models to guide the implementation of low-level designs. By symbolically executing the device traces collected in Section 3.3 on the corresponding FDMs, our approach enables the consistency and correctness checking between different levels of SoC designs (i.e., virtual prototyping, post-silicon design).

3.4.1 Formal Definitions. In both FDMs and silicon devices we consider two kinds of registers, i.e., interface registers that explicitly reflect the interaction between hardware and software components, and internal registers that are not observable. Based on these registers, Definition 3.3 presents the formal definition of an FDM state.

DEFINITION 3.3. *An FDM state is denoted as $F = \{F_I, F_N\}$, where F_I and F_N indicate the assignments to its interface register set R_I and internal register set R_N, respectively.* ■

When employing symbolic execution, a **concrete FDM/device state** is a state whose register values are all concrete, and a **symbolic FDM/device state** is a state whose registers are assigned with symbolic values and there can be constraints on these symbolic values. As an abstraction, a symbolic FDM/device state can be considered as a set of concrete symbolic FDM/device states. In our approach, the state of an FDM F and the state of a virtual/silicon device S are all treated as symbolic states. We use $symb(F)$ and $symb(S)$ to denote the two sets of concrete states for F and S, respectively. Definition 3.4 defines the conformance between them.

DEFINITION 3.4. *An FDM state F and a virtual/silicon device state S conform to each other if $symb(F) \cap symb(S) \neq \emptyset$.* ■

Let $v_1, v_2, \ldots, v_n$ denote the state variables that correspond to the registers used in F and S. We construct the expression A_S as $(v_1 == V_{1\_S}) \wedge (v_2 == V_{2\_S}) \wedge \ldots \wedge (v_n == V_{n\_S})$ to indicate the state S, where $(v_i == V_{i\_S})$ means that the value of the i^{th} register in S is Val_{i_S}. Similarly, we construct the expression A_F as $(v_1 == V_{1\_F}) \wedge (v_2 == V_{2\_F}) \wedge \ldots \wedge (v_n == V_{n\_F})$ to indicate the state F. Since some registers in F may have symbolic values, we use C_F to denote the constraint of F. Given A_S, A_F and C_F, $symb(F) \cap symb(V) \neq \emptyset$ holds if and only if $A_S \wedge A_F \wedge C_F$ is satisfiable.

3.4.2 Implementations. Our approach checks conformance between high-level FDMs and low-level virtual/silicon devices based on the comparison of corresponding states of their traces triggered by the same driver requests. However, due to the hardware concurrency of an SoC design, it is hard to check all its possible behaviors. To simplify the consistency checking, we symbolically execute the device traces using the harness module (see Listing 1) on FDMs in a non-deterministic manner. For each driver request, we check the device state with all the possible FDM states using the condition presented in Definition 3.4.

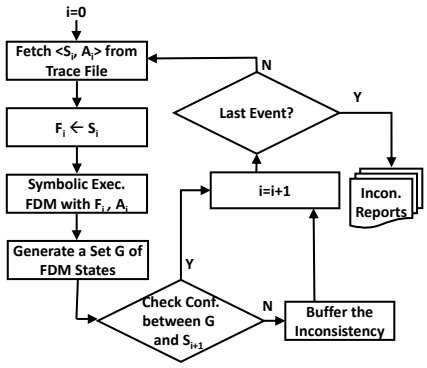


Figure 3: Procedure of Conformance Checking

Figure 3 presents a detailed procedure of our conformance checking approach between FDMs and virtual/silicon devices. In this procedure, we use i to denote the index of driver requests. Firstly, the conformance checker fetches the state-action pair <S_i,A_i> from the beginning of the device trace file, and assigns the device state S_i to the current state of FDM, i.e., F_i. Triggered by the driver request A_i, the FDM is symbolically executed from the state F_i. Upon the completion of the symbolic execution of A_i on FDM, we collect all the possible resultant states and save them in a set G. Then we check the consistency between the FDM and the device using the condition defined in Definition 3.5. If it is not satisfied, the identified inconsistency will be saved. For each inconsistency, our conformance checker will record the following three information: i) inconsistent state (i.e., S_{i+1}), ii) triggering driver request (i.e., A_i), and iii) an FDM execution path under this driver request. Note that the inconsistency does not abort the whole procedure. Even if an inconsistency is detected, the procedure will continue the symbolic execution based on the latest device state in the trace until all

the device states are traversed. When the procedure finishes, if there exists any inconsistency in the buffer, all the saved inconsistencies will be reported. Note that the original version of KLEE does not accept the hardware traces as inputs. Moreover, it does not support the conformance checking between different SoC design layers. To enable the conformance checking procedure as shown in Figure 3, we modified the KLEE code and included our trace execution and conformance checking methods.

DEFINITION 3.5. *Let $G = \{g_k \mid 0 \leq k \leq m\}$ be the set of FDM states. Let F_{i+1} be the next device state triggered by A_i. The FDM and the virtual/silicon device conform to each other at A_i if $\exists g_j \in G$ where $0 \leq j \leq m$, $symb(F_{i+1}) \cap symb(g_i) \neq \emptyset$.* ■

4 PERFORMANCE EVALUATION

To evaluate the effectiveness of our approach, we applied our approach to two industrial network adapters (i.e., Intel *e1000* and Intel *eepro100*), which have both virtual and silicon versions. The virtual devices used in this experiment are all available from QEMU (version 0.15.1). All the experimental results were obtained on a Ubuntu desktop with AMD 3.2GHz processors and 16GB RAM.

Table 1: Experimental Settings for Network Adapters

Devices	Spec. (LoC)	FDM (LoC)	VP (LoC)	Select. Captured Size (Bytes)
Intel e1000 Gigabit NIC	546	1805	2099	1224
Intel eepro100 Megabit NIC	587	903	2178	74

We developed the extended SystemRDL specifications for the both network adapters according to their software developer's manuals. Since our extended SystemRDL specifications are considered as high-level abstractions, they only take partial interface registers of the investigated network adapters into account. By using our FDM generator, we can obtain corresponding executable FDMs in less than 0.1 seconds. Table 1 presents the experimental setting information. The first column gives basic descriptions of the devices. For each network adapter, columns 2-4 present the Lines of Code (LoC) information of the extended SystemRDL, executable FDM, and virtual prototype (from QEMU), respectively. The last column gives the address scope size of selected registers, which are captured by our trace recorder. Note that under the stringent time-to-market requirement our FDM-based approach is more suitable for conformance checking than virtual prototyping. This is because our specification-driven approach needs much smaller manual efforts than the ones devoted to virtual prototyping.

4.1 Identified Inconsistencies and Bugs

To check whether the behaviors of virtual prototypes and silicon devices are consistent to the generated FDM, we applied the test cases shown in Table 2 on both virtual and silicon devices, and collected their traces. Each trace consists of a sequence of driver requests triggered by some command. We considered four types of network commands which are frequently used daily. For example, when investigating "Transfer files" at virtual prototype and silicon layers, we ran the command *scp* to copy a 2.5GB file from some server via internet on both QEMU and the desktop with virtual/silicon network adapters, respectively.

Table 2: Test Cases for Evaluation

Types	Test Cases	Descriptions
Reset Network Interface	ifdown	Bring down the network interface
	ifup	Bring up the network interface
Ping	ping	Send ICMP ECHO_REQUEST
Transfer files	scp	Copy a 2.5GB file from some network server
NIC test-suite	ifconfig	Configure a network interface
	hping3	Manipulate network packets

For each network adaptor design, we symbolically executed its FDM using the traces collected from virtual prototypes and silicon devices, respectively. During the symbolic execution of a trace our approach executes the FDM driver requests one by one. One inconsistency here means that the resultant interface register state of a virtual or silicon device triggered by some driver request is not consistent with the interface register state of the given FDM. Note that instead of aborting the symbolic execution when encountering one inconsistency, our approach continues to execute the remaining traces starting from the inconsistent register state. Table 3 presents the 12 bugs caused by inconsistencies for both virtual and silicon devices by using the test cases shown in Table 2. We classified these bugs into six types, where each type may have multiple bugs. The indices and type descriptions of the bugs are presented in the columns 1-2 of Table 3. Note that each bug type may have different kinds of bugs. For example, since for the bug type E1 there are three different reserved registers of silicon devices that are updated by mistake, we consider them as three different bugs. We use column 3 to denote the number of bugs of specific bug types. The last column shows the types of devices where the bugs are located. All the above bugs may disrupt the driver executions on virtual/silicon devices, resulting in serious problems such as system crash.

Table 3: Bugs Identified from Virtual and Silicon Devices

Indices	Bug Types	Num.	Bug Sources
E1	Update the bits of reserved SD register	3	SD
E2	Generate unnecessary interrupts	1	VP
E3	Fail to update register when necessary	2	VP
E4	Write incorrect values to registers	3	VP
E5	Update the bits of reserved VP register	1	VP
E6	Driver issues a write to reserved registers	2	Driver

* VP and SD stand for Virtual Prototype and Silicon Device, respectively.

We compared our approach with the virtual prototype based conformance checking approach proposed in [7]. For a fair comparison, we use the same test cases as shown in Table 2. Table 4 shows the comparison result. In the last three columns, we present the number of bugs identified using different conformance checking methods. We use *X-Y* to denote the conformance checking between the two layers of *X* and *Y*, where *X* is symbolically executed using the traces collected from *Y*. Since FDM is the golden reference model, here we only consider two kinds of bugs, i.e., virtual device bugs and silicon device bugs.

Table 4: Comparison of Different Methods

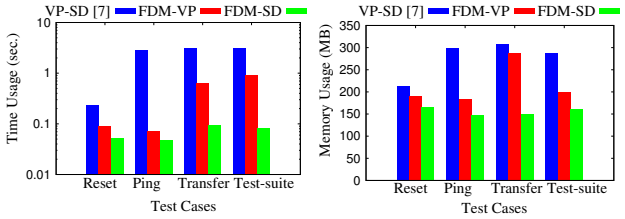
Bug Source	Bug Type	FDM-VP	FDM-SD	VP-SD [7]
Silicon Devices	E1	-	3	2
Virtual Devices	E2	1	-	1
	E3	2	-	2
	E4	3	-	3
	E5	1	-	-
Driver	**E6**	1	1	-

From Table 4, we can find that our *FDM-SD* approach can identify one more SD bugs that cannot be found by the *VP-SD* method presented in [7]. We find that our *FDM-VP* approach can cover all the bugs identified by the *VP-SD* method. In addition, our approach can find two new types of bugs, i.e., E5 and E6. For the bug of type E5, we found an incorrect implementation of the virtual device, which can update some reserved registers. For the bugs of type E6, we found two inconsistencies from both virtual and silicon device traces. Based on the reported results, we find that bugs are not located in virtual or silicon devices. Instead, the bugs come from the device drivers, which can write reserved device registers by mistake. Note that the reason why the bugs of types E5 and E6 cannot be identified by the *VP-SD* method is mainly due to the lack of golden reference models

(i.e., FDMs). Without a formal specification, the improper usage of reserved VP registers is hard to be discovered by the *VP-SD* approach. By detecting inconsistencies using our FDM-based approach, we can easily identify design bugs and figure out why the drivers cannot work properly with virtual/silicon devices.

4.2 Efficiency of Conformance Checking

We evaluated the efficiency of our approach in terms of time usage, memory usage and false positive/negative ratios. All the results are obtained using the test cases shown in Table 2.



(a) Time Usage for *e1000 NIC* (b) Memory Usage for *e1000 NIC*

Figure 4: Comparison of Time and Memory Usages

Figure 4 shows the comparison of time and memory usages for *e1000 NIC* design between our FDM-based approaches and the *VP-SD* method proposed by [7]. Note that since the amount of driver requests is hard to control, to facilitate the evaluation we only present the average time usages for one driver request in Figure 4(a). We can find that our *FDM-VP* and *FDM-SD* methods can achieve up to 67 times improvement on time compared with *VP-SD*. This is mainly because our FDM model is more abstract than its corresponding virtual prototype. Figure 4(b) shows the maximum memory usage when running different test cases. It can be observed that our approaches need much less memory for the conformance checking compared with the *VP-SD* method. The reason for these benefits is that our FDM-based approaches abstract all the internal operations of a design which are irrelevant to interface registers, while the virtual prototype-based method [7] needs to consider all the internal operation details. For example, upon receiving a packet, the FDM of some network adapter only updates the value of interface registers without dealing with internal states (e.g., sorting packets in buffers). However, virtual prototypes need to take all these internal operations into consideration.

Table 5: Number of Discovered/Verified Inconsistencies

Devices	Test Cases	Discovered/Verified		
		VP-SD [7]	FDM-VP	FDM-SD
Intel e1000 Gigabit NIC	Reset NIC	8/8	14/14	**5/3**
	Ping	8/8	8/8	2/2
	Transfer files	**12/9**	14/14	**4/2**
	Test-suite	11/11	6/6	3/3
Intel eepro100 Megabit NIC	Reset NIC	4/4	11/11	0/0
	Ping	2/2	8/8	0/0
	Transfer files	2/2	6/6	0/0
	Test-suite	4/4	8/8	0/0

The false positive/negative ratios can be used to judge the capabilities of different conformance checking methods. From Table 4, we can observe that our approach outperforms the *VP-SD* method [7] from the perspective of false negatives. Since our approach can detect more bugs than *VP-SD*, it can achieve better false negative ratio accordingly. To investigate the false positive ratio, Table 5 presents the statistics of both discovered and verified "inconsistencies" when executing various test cases with different approaches. Note that the numbers in columns 3-5 denote the inconsistencies identified during the testing, where some inconsistencies may be caused by the same bug. We marked the false positives as bold in the table. When the *FDM-VP* approach is used, we can find that there is no false positives. In other words, *FDM-VP* can accurately identify the inconsistencies for the virtual prototypes of network adapters. When using *FDM-SD* for conformance checking, we can identify fewer inconsistencies during the execution of silicon devices. However, for the *e1000 NIC* design there are two test cases resulting in false positives. Based on trace analysis, we found that the false positives are caused by the incomplete modeling of registers. Note that interface registers defined in the extended SystemRDL specifications only cover the registers that interest driver designers. However, during the silicon execution the values of some modeled registers can be modified by other registers that are not defined in SystemRDL. When some driver request triggers this scenario, the new silicon state reflects the modification, but the new FDM state does not. Although an inconsistency is reported in this case, it does not mean that the FDM is incorrect. By adding some register access behaviors in the extended SystemRDL specification, all the four false positives can be eliminated.

5 CONCLUSION

Although there exist various validation approaches for virtual prototype and post-silicon designs in the top-down SoC design flow, it is still difficult to ensure that post-silicon designs are correctly implemented as required. The main reason is due to the lack of high-level formal models that can accurately describe the device behaviors and be used as the golden reference model for the subsequent design refinement. In this paper, we presented a novel specification-driven approach that can automatically conduct the conformance checking for both virtual prototype and post-silicon designs. Experimental results show that our approach can effectively identify real bugs from both virtual devices excerpted from QEMU [6] and corresponding industrial silicon devices, some of which cannot be discovered by state-of-the-art approaches.

ACKNOWLEDGMENTS

This research was partially supported by National Science Foundation of China (Grant No. 61672230), Shanghai Municipal National Science Foundation (Grant No. 16ZR1409000), National Science Foundation (Grant No. CNS-1422067) and gifts from Intel Corporation. Mingsong Chen is the corresponding author.

REFERENCES

[1] M. J. Renzelmann, A. Kadav, and M. M. Swift. SymDrive: Testing drivers without devices. In *Proc. of OSDI*, 279–292, 2012.
[2] S. Nelson and P. Waskiewicz. Virtualization: Writing (and testing) device drivers without hardware. In *Proc. of Linux Plumbers Conference*, 2011.
[3] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proc. of OSDI*, 1–16, 2004.
[4] 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.
[5] SystemRDL 1.0 Standard. http://www.accellera.org/downloads/standards /systemrdl.
[6] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of ATC*, 41–46, 2005.
[7] L. Lei, F. Xie, and K. Cong. Post-silicon conformance checking with virtual prototypes. In *Proc. of DAC*, 29:1–29:6, 2013.
[8] N. Bombieri, F. Fummi, G. Pravadelli, and J. Marques-Silva. Towards equivalence checking between TLM and RTL models. In *Proc. of MEMOCODE*, 113–122, 2007.
[9] M. Chen and P. Mishra. Assertion-based functional consistency checking between TLM and RTL models. In *Proc. of VLSI Design*, 320–325, 2013.
[10] P. Herber, M. Pockrandt, and S. Glesner. Automated conformance evaluation of SystemC designs using timed automata. In *Proc. of ETS*, 188–193, 2010.
[11] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40-44, 2012.
[12] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proc. of ASPLOS*, 265–278, 2011.
[13] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of OSDI*, 209–224, 2008.
[14] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. of ESEC/FSE*, 263–272, 2005.
[15] X. Qin and P. Mishra. Scalable test generation by interleaving concrete and symbolic execution. In *Proc. of VLSI Design*, 104–109, 2014.