AUSTRALIAN JOURNAL OF BASIC AND APPLIED SCIENCES

ISSN:1991-8178 EISSN: 2309-8414
Journal home page: www.ajbasweb.com

Implementing Selective Symbolic Execution for Regressive Testing Virtual Prototype

[1]S. K. HariKarthik, [2]Dr. V. Palanisamy, [3]Dr. P. Ramanathan

[1]Assistant Professor, Dept. of Information Technology, INFO Institute of Engineering, Coimbatore, India.
[2]Principal[Rtd], INFO Institute of Engineering, Coimbatore, India.
[3]Professor & Head, Dept of ECE, INFO Institute of Engineering, Coimbatore, India.

Address For Correspondence:
S. K. HariKarthik, Assistant Professor, Dept. of Information Technology, INFO Institute of Engineering, Coimbatore, India.

ARTICLE INFO

Article history:
Received 26 April 2016
Accepted 21 July 2016
Published 30 July 2016

Keywords:
Regression Testing, Virtual Prototypes, Virtual Platform, Selective Symbolic Execution, Symbolic Execution.

ABSTRACT

Virtual prototype is widely used in all electronics based software development companies. This will make the software development processes more effective and efficient. For virtual prototyping, testing is one of the major problems. To overcome that problem we introduce selective symbolic execution for regressive testing. Prototype is developed and released in different versions to overcome the defect of old version; the changes are tested by using regressive test methodology. For the first version symbolic execution can be used to execute all the paths, to find the bugs. In the next version using symbolic execution will take more time to execute all the paths. To overcome this problem we introduce the selective symbolic execution, which will test only the symbols which had undergone changes.

INTRODUCTION

Nowadays there has been great pressure on electronics product developers to shorten the time-to-market and improve the product quality. However, time-to-market and product quality are usually opposing attributes in a product development process. Developers can shorten the time -to-market by skipping validation steps, thus it can harm the product quality. On the other hand, if developers want to improve product quality, more validation effort need to be devoted which means more time is required. This demands innovative approaches and efficient methodologies to accelerate product development and validation to save time and improve quality (Bin Lin, and Dejun Qian, 2016).

Symbolic execution is a powerful technique for analyzing program behavior, finding bugs, and generating tests, but suffers from severely limited scalability: the largest programs that can be symbolically executed today are on the order of thousands of lines of code. To ensure feasibility of symbolic execution, even small programs must curtail their interactions with libraries, the operating system, and hardware devices. This paper introduces selective symbolic execution, a technique for creating the illusion of full system symbolic execution, while symbolically running only the code that is of interest to the developer. A prototype that can symbolically execute arbitrary portions of a full system, including applications, libraries, operating system, and device drivers. It seamlessly transitions back and forth between symbolic and concrete execution, while transparently converting system state from symbolic to concrete and back. Our technique makes symbolic execution practical for large software that runs in real environments, without requiring explicit modeling of these environments.

Recently virtualization and virtual prototyping techniques have been widely used in electronics companies, such as Intel and ARM, to accelerate the product development cycle (Eschweiler, D. and V. Lindenstruth, 2015;

To Cite This Article: S. K. HariKarthik, Dr. V. Palanisamy, Dr. P. Ramanathan., Implementing Selective Symbolic Execution for Regressive Testing Virtual Prototype. ***Aust. J. Basic & Appl. Sci.,*** *10(12): 110-116, 2016*

Mueller, W., *et al*., 2012). These techniques provide a convenient way for developers to start software development without silicon prototypes (Nelson, S. and P. Waskiewicz, 2011). As shown in Figure 1, in the traditional product development process, software development has waited until the first RTL design or FPGA prototype becomes available. Since Virtual Prototypes (VP) are high-level functional models, the VP development requires less effort and can be delivered to software developers much earlier. With the Virtual Prototypes and virtual platform, software developers can start driver and firmware development much earlier than before. In this way, product development team can shift-left product development process and reduce the overall time. Moreover, virtual prototypes can be used as reference models for generating post-silicon functional test cases (Cong, K., *et al*., 2013) and checking the correctness of physical devices (Lei, L., *et al*., 2013). Furthermore, virtual prototypes can be mixed with emulation and FPGA platform for early system integration and hardware/software validation (Srinivasan, V., *et al*., 2015; Li, H., *et al*., 2010).

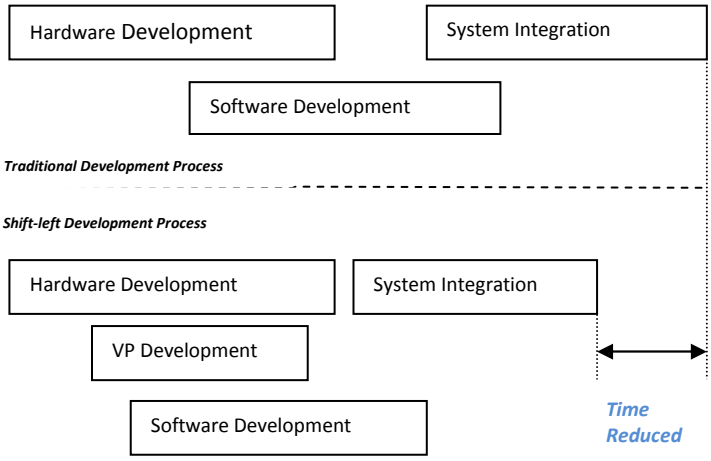

Fig. 1: Traditional development VS VP-based shift-left development

Since virtual prototypes are software models developed according to the hardware specifications by developers, it is very important to validate virtual prototypes. Virtual prototypes change frequently due to feature updates, specification changes and bug fixes. To validate the changes, developers need to perform regression testing on virtual prototypes to ensure that the changes have not introduced new faults. However, applying traditional regression testing to validating virtual prototypes is difficult. This demands a new approach to regression testing of virtual prototypes.

Symbolic execution, selective symbolic execution and concolic execution (Vitaly Chipounov; Cadar, C., *et al*., 2008; Chipounov, V., *et al*., 2011; Kannavara, R., *et al*., 2015; Godefroid, P., *et al*., 2012; Kim, M., *et al*., 2012; Marinescu, P. and C. Cadar, 2012) have been widely used for validating software programs. Many symbolic execution tools and related techniques have been developed and used for validating software programs and detecting security issues (Chaudhuri, A. and J. Foster, 2010; Cho, C., *et al*., 2011). In the past several years, symbolic execution has been further applied to hardware domain (Cong, K., 2015l Li, G., *et al*., 2012; Liu, L., 2014). Symbolic execution of RTL design can be used for validating RTL designs [29, 30] and generating high-quality test vectors for design testing (Yang, Z., *et al*., 2013; Yang, Z., *et al*., 2014; Qin, X. and P. Mishra, 2014). Symbolic execution of virtual prototypes has been deeply explored and utilized for coverage analysis (Cong, K., *et al*., 2014), test generation (Cong, K., *et al*., 2013) and conformance checking for post-silicon functional validation (Lei, L., *et al*., 2013; Lei, L., *et al*., 2013; Lei, L., *et al*., 2014).

In this paper, we propose a new regression testing approach for checking conformance between two versions of virtual prototypes. Our approach takes the old version of virtual prototype as the reference model and executes it symbolically to collect all possible path information. For each path explored, the new version of virtual prototype is executed following the path constraints collected by using selective symbolic execution methodology. Final result is compared between the old and new versions to check if both versions conform. We have applied using selective symbolic execution to retest updated software, result was compared it is effective and efficient.

Our research makes four main contributions as follows:

A. Create a regression testing test case for checking old versions and new version of a virtual prototype:

A test case is proposed for regression testing of virtual prototypes. The test case takes old versions and new version of a virtual prototype as inputs. Then symbolic execution is conducted for first version (Bin Lin, and

Dejun Qian, 2016) and selective symbolic execution for new version. Final device states are collected to identify the differences between two versions.

B. Generate a test harness for guiding symbolic execution and collecting result:

In order to guide symbolic execution, we need to create a test harness to connect old versions of virtual prototypes together. Moreover, the developers can decide what device states they want to check in the test harness (Bin Lin, and Dejun Qian, 2016).

C. Generate a test harness for guiding Selective symbolic execution, collecting information and compare with symbolic execution result:

In order to guide selective symbolic execution, we need to create a test harness to connect new version of virtual prototypes. The result are collected and compared with the result of old version collected using symbolic execution.

D. Evaluate on a widely-used virtual prototype:

We have evaluated our approach on a QEMU E1000 virtual network device. The experimental results show that our approach can efficiently detect differences between two different versions (Bin Lin, and Dejun Qian, 2016).

The remainder of this paper is structured as follows. Section 2 compares differences between a silicon device and the corresponding virtual prototype (Bin Lin, and Dejun Qian, 2016). Section 3 presents our approach. Section 4 demonstrates the experimental results. Section 5 discusses the related work. Section 6 concludes and discusses future work.

II. Silicon Device Vs Virtual Prototype:

A virtual prototype is a software model which emulates necessary behaviors defined in the hardware specification. A virtual prototype should behave the same as the corresponding silicon device from the view of software developers. In order to better introduce what a virtual prototype is, we compare the differences between a silicon device and the corresponding virtual prototype. In the following, we use PCI devices as examples since PCI devices are complex and widely used.

A. Silicon Device:

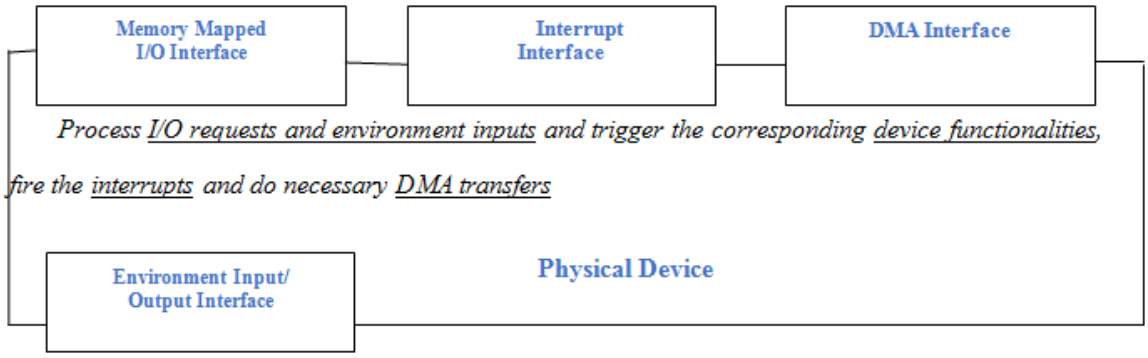

Fig. 2: An overview of a physical device

As shown in Figure 2, a physical device includes two parts: interfaces and internal functionalities. A PCI physical device commonly includes four interfaces:

1) *Memory Mapped I/O Interface:* the CPU performs write/read register operations on the device.
2) *Interrupt Interface:* the device sends electronic signals to the CPU notifying a hardware event.
3) *DMA Interface:* the device accesses the main system memory independently of the CPU.
4) *Environment Input/output Interface:* the device sends the output data to the environment and the environment sends the input data to the device.

Inside a silicon device, all device functionalities are implemented as electronic logic to process I/O requests and environment inputs and trigger the corresponding device functionalities, fire interrupts and do necessary DMA transfers.

A silicon device is connected to the system board through the system bus. The silicon device is also connected to outside environment through different connections, such as network cables and VGA connectors.

B. Virtual Prototypes:

As shown in Figure 3, a virtual prototype also includes two parts: interface functions and internal behavioral functions. A PCI virtual prototype includes four kinds of interface functions:

1) *Memory Mapped I/O Function:* when there is a register write/read CPU operation, the corresponding interface functions are called to process the requests.
2) *Interrupt Function:* when a virtual prototype needs to fire an interrupt, the interrupt function is called to notify the virtual platform.
3) *DMA Function:* the device accesses the main system memory through DMA interface functions.

4) *Environment Input/output Function*: the output function is used by the virtual prototype to send data while the input function is invoked by the virtual platform to notify the virtual prototype there is data received.

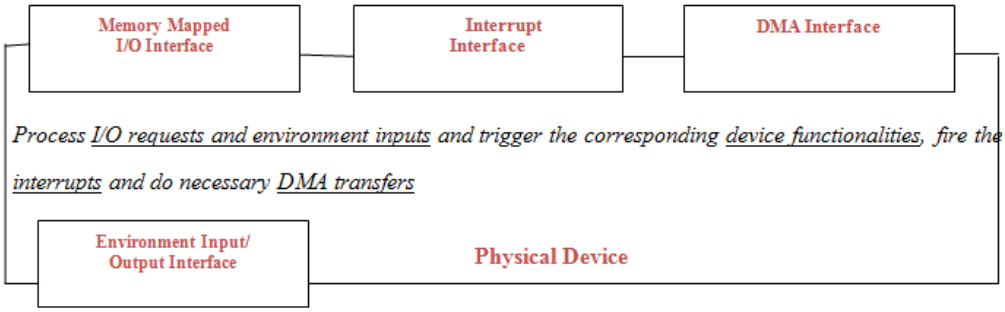

Fig. 3: An overview of a virtual prototype

Inside a virtual prototype, all device functionalities are implemented as software functions to process I/O requests and environment inputs and trigger the corresponding device functionalities, fire interrupts and do necessary DMA transfers.

A virtual prototype is one component of a virtual platform. The memory mapped I/O functions and environment input functions are defined in the virtual prototype and invoked by the virtual platform to perform register access and process received data. The interrupt functions, DMA functions and environment output functions are defined in the virtual platform and called in the virtual prototype to fire interrupts, perform DMA access and send the data to the environment.

III. Our Approach:

A. Overview:

Virtual prototypes change frequently due to feature updates, specification changes and bug fixes. To validate the changes, developers need to perform regression testing on virtual prototypes to ensure that the changes have not introduced new faults [1]. The basic idea of our approach is to run fist version of virtual prototype by using symbolic execution method, it shows the bugs arise in the first version. Then the new version is developed to by fixing the bugs of first version. So it's no need to test the entire coding by using symbolic execution. Instead of that, we execute the selective symbolic method to test the new version. Here we test only the code which undergo the changes.

Suppose we have two versions of a virtual prototype V_{old} and V_{new}, our approach can efficiently detect V_{old} by symbolic execution and V_{new} by selective symbolic execution method. The basic workflow of our approach is shown in Figure 4.

Our approach takes two versions of a virtual prototypes as inputs. First, the old version of the virtual prototype is executed symbolically and all paths are explored. For each path explored, the path constraints *C* and the final device state *S* are collected and defects are identified. After rectifying the defect we use selected symbolic execution method for new version. It is executed selective symbolically and the path constraints *C'* and the final device state *S'* for each path are collected. The last step of our approach is equivalence checking. With the path constraints *C'*, we compare *S* and *S'* to detect the differences between V_{old} and V_{new}.

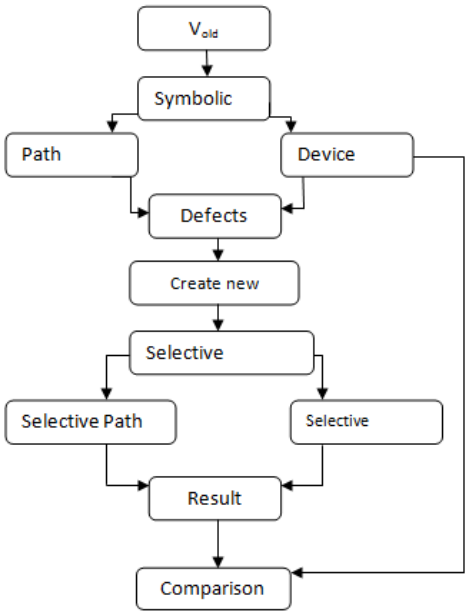

Fig. 4: The workflow of our approach

B. Test Harness Generation:

A virtual prototype is a software model which is not a standalone program. To apply symbolic execution approach to regression testing of virtual prototypes, a test harness is needed to compose a complete program. The test harness mainly includes two parts:

1) To invoke the interface functions of a virtual prototype correctly, need to construct a device state variable and device request variable. Moreover, interface functions should be called to trigger device functionalities under a desired device state upon a device request (Bin Lin, and Dejun Qian, 2016).

2) To conduct symbolic execution and equivalence checking, we need to make necessary variables as symbolic variables and invoke some special functions to guide execution and collect runtime constraints and device states (Bin Lin, and Dejun Qian, 2016).

In our case study, we use QEMU E1000 virtual prototype (Bellard, F., 2005; QEMU, 2015). An excerpt of the test harness we generated

C. Symbolic Execution:

In the paper (Bin Lin, and Dejun Qian, 2016), they have demonstrated how to conduct symbolic execution of virtual prototypes for regression testing. Symbolic execution engine is further modified KLEE for our specific regression testing approach for old version.

D. Selective Symbolic Execution:

1) To construct device states for both versions of a virtual prototype and assign the selective symbolic value to new version

2) Implementations to save device states and register read return values after invoking the changed code interface states.

3) Ability to compare two device states that recoded in selective symbolic execution and symbolic execution. After comparing device states, a final report is provided for further analysis.

4) Removed the paths that are not undergone changes in new version.

IV.A Case Study:

A. Overview:

We have applied our approach to a QEMU E1000 virtual network prototype as mentioned in paper (Bin Lin, and Dejun Qian, 2016). All experiments have been conducted on a physical machine with 2.5GHz CPU and 4GB memory.

In our case study, we verified if two versions of the E1000 virtual prototype have the same final states after processing external requests. Such external requests include MMIO mapped register read and write, network packet receive request. In our test harness, the corresponding interface functions are invoked separately to trigger different device functionalities and conduct difference checking (Bin Lin, and Dejun Qian, 2016).

To apply our approach as paper (Bin Lin, and Dejun Qian, 2016), we have selected two different versions of QEMU release: 0.13.0, 1.3. In Table 1. These 2 versions were released on different years and have different

sizes.

Table 1: Summary of two different versions

Version	LOC	Release	Date
0.13.0	969	Nov 29,	2010
1.3.1	1105	Jan 28, 2013	

B. Detected Differences:

After applying paper (Bin Lin, and Dejun Qian, 2016) approach to old version 0.13.0 our approach to new versions 1.3.1, most possible paths were explored and many differences were detected. .

Table 2: Summary of detected differences

	KLEE		S2E		
	Coverage	Min	Coverage	Min	increase
Interrupt	66%	40	74%	8	+15%
DMA	87%	90	89%	22	+5%
I/O function	86%	89	89%	20	+5%

As shown in Table 2, percentage of paths covered with two methods and coverage time is also mentioned. In our approach the percentage of coverage is increased.

V. Related work:

Virtual platforms and virtual prototypes have been more and more utilized by different electronic vendors and academic research groups. There are many open-source and commercial virtual platforms like Simics (Wind River, 2015) and QEMU (2015) developed in the past decade (Wikipedia, 2015). Different virtual platform solutions have been developed by three large electronic design automation companies, Synopsys, Cadence and Mentor Graphics (Wind River, 2015; Wikipedia, 2015; Synopsys, 2015). Under different virtual platforms, many virtual prototypes have been developed and utilized. These virtual prototypes cover different kinds of hardware devices, such as network, USB, audio and video.

Conclusions:

In this paper, we present how to apply comparison of symbolic execution and selective symbolic execution execute for different versions of a virtual prototype for regression testing. We have detected many differences between two versions of the QEMU E1000 virtual prototype. The experimental results show that our approach can efficiently work for the regression testing which is tested repeatedly for each version after rectifying the defects in older version. In the future, we will apply our approach to more virtual prototypes and test case prioritization also can applied to the new versions.

REFERENCES

Bin Lin, and Dejun Qian, 2016. “Regression Testing of Virtual Prototypes Using Symbolic Execution”in arxive cornel university library.

Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, George Candea, “Selective Symbolic Execution” http://dslab.epfl.ch/pubs/selsymbex.pdf.

Eschweiler, D. and V. Lindenstruth, 2015. "Test driven development for device drivers and rapid hardware prototyping," in IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW).

Mueller, W., M. Becker, A. Elfeky and A. DiPasquale, 2012. "Virtual prototyping of Cyber-Physical Systems," in ASP-DAC.

Nelson, S. and P. Waskiewicz, 2011. “Virtualization: Writing (and testing) device drivers without hardware,” in Linux Plumbers Conference.

Cong, K., F. Xie and L. Lei, 2013. “Automatic concolic test generation with virtual prototypes for post-silicon validation,” in ICCAD.

Lei, L., F. Xie and K. Cong, 2013. “Post-silicon conformance checking with virtual prototypes,” in DAC.

Srinivasan, V., F. Schirrmeister, V. Singh and R. Klein, 2015. “Why hybrid platforms are needed for pre-silicon hardware and software development,” in EDPS.

Li, H., D. Tong, K. Huang and X. Cheng, 2010. “Femu: A firmware-based emulation framework for soc verification,” in CODES+ISSS.

Bellard, F., 2005. “QEMU, a fast and portable dynamic translator,” in USENIX ATEC.

QEMU, 2015. “QEMU,” http://wiki.qemu.org/Main.

Cong, K., F. Xie and L. Lei, 2013. “Symbolic execution of virtual devices,” in QSIC.

Cadar, C., D. Dunbar and D. Engler, 2008. “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” in OSDI.

Wind River, 2015. “Simics full system simulator,” tp://www.windriver.com/products/simics.

Wikipedia, 2015. “Comparison of platform virtualization software”, https://en.wikipedia.org/wiki/Comparison_of_platform_virtualization_software.

Synopsys, 2015. “Synopsys virtual prototyping solutions,” http://www.synopsys.com/prototyping/virtualpro otyping/Pages/default.aspx.

Cadence, 2015. “Cadence virtual system platform,” http://www.cadence.com/products/sd/virtual_syste m/pages/default.aspx.

Mentor Graphics, 2015. “Vista Virtual Prototyping for SystemC”, https://www.mentor.com/esl/vista/virtual- prototyping.

Chipounov, V., V. Kuznetsov and G. Candea, 2011. “S2E: a platform for in-vivo multi-path analysis of software systems”, in ASPLOS.

Kannavara, R., C. Havlicek, B. Chen, M. Tuttle, K. Cong, S. Ray and F. Xie, 2015. “Challenges and Opportunities with Concolic Testing”, in IEEE National Aerospace Conference / Ohio Innovation Summit & IEEE Symposium on Monitoring & Surveillance Research (NAECON-OIS).

Godefroid, P., M. Levin and D. Molnar, 2012. “SAGE: Whitebox Fuzzing for Security Testing”, Queue 10.

Kim, M., Y. Kim and G. Rothermel, 2012. "A Scalable Distributed Concolic Testing Approach: An Empirical Evaluation," in Software Testing, Verification and Validation (ICST).

Marinescu, P. and C. Cadar, 2012. “make test-zesti: a symbolic execution solution for improving regression testing”, in ICSE.

Chaudhuri, A. and J. Foster, 2010. “Symbolic security analysis of ruby-on-rails web applications”, in CCS.

Cho, C., D. Babić, P. Poosankam, K. Chen, E. Wu and D. Song, 2011.“MACE: model-inference-assisted concolic exploration for protocol and vulnerability discovery”, in SEC.

Cong, K., 2015. “Post-silicon functional validation with virtual prototypes,” Ph.D. dissertation, Portland State University.

Li, G., P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh and S. Rajan, 2012. “GKLEE: concolic verification and test generation for GPUs”, in PpoPP.

Liu, L., 2014. “Harmonizing data mining and static analysis to tackle hardware and system level verification”, Ph.D. dissertation, University of Illinois at Urbana- Champaign.

Yang, Z., K. Hao, K. Cong, S. Ray and F. Xie, 2013. “Equivalence Checking for Compiler Transformations in Behavioral Synthesis”, in ICCD.

Yang, Z., K. Hao, K. Cong, L. Lei, S. Ray and F. Xie, 2014. “Scalable Certification Framework for Behavioral Synthesis Front-End”, in DAC.

Qin, X. and P. Mishra, 2014. "Scalable Test Generation by Interleaving Concrete and Symbolic Execution," in 27th International Conference on VLSI Design and 13th International Conference on Embedded Systems.

Liu, L. and S. Vasudevan, 2014. “Scaling Input Stimulus Generation through Hybrid Static and Dynamic Analysis of RTL”, ACM Trans. Des. Autom. Electron. Syst.

Liu, L. and S. Vasudevan, 2011. "Efficient validation input generation in RTL by hybridized source code analysis," in DATE.

Cong, K., L. Lei, Z. Yang and F. Xie, 2014. “Coverage Evaluation of Post-silicon Validation Tests with Virtual Prototypes”, in DATE.

Lei, L., K. Cong and F. Xie, 2013. “Optimizing Post-silicon Conformance Checking”, in ICCD.

Lei, L., K. Cong, Z. Yang and F. Xie, 2014. “Validating Direct Memory Access Interfaces with Conformance Checking”, in ICCAD.