Symbolic Quick Error Detection for Pre-Silicon and Post-Silicon Validation: Frequently Asked Questions

Eshan Singh, David Lin, Clark Barrett, and Subhasish Mitra
Stanford University

Editor's notes:
Reducing the error detection latency is critical for improving the design visibility while searching for design errors. This article uses a FAQ format to discuss the key points of the symbolic QED method that can be applied during both pre-silicon and post-silicon validation.

—Nicola Nicolici, McMaster University and Haralampos-G. Stratigopoulos, Sorbonne University

Question 1: What are major pre-silicon verification challenges?

Pre-silicon verification checks whether the implemented design meets defined specifications. Both simulation-based techniques and formal methods are used at different levels of abstraction, from the architectural level to gate-level netlists. Current verification approaches, however, are too slow and simply unable to handle the growing size and complexity of Integrated Circuits (ICs). Simulation is several orders of magnitude slower than actual silicon. Emulation and Field-Programmable Gate Array (FPGA) prototyping offer faster performance, but struggle to scale for large designs. Formal verification can be applied in certain situations, e.g., individual arithmetic units or protocols, but faces similar scalability challenges for full chip-level verification. As a result, "difficult" and often critical design flaws (bugs) frequently escape pre-silicon verification. Design bugs can be broadly classified as follows:

1) logic bugs that are caused by (logic) design errors;
2) electrical bugs that are caused by interactions between a design and its "electrical" state.

Existing pre-silicon verification is insufficient for "difficult" logic bugs. It also does not adequately address electrical bugs that appear only after ICs are manufactured.

Question 2: Why is post-silicon validation important and what challenges does it face?

The growing number of design bugs that escape pre-silicon verification means that there is an increasing dependence on post-silicon validation of manufactured ICs in actual system environments to detect and fix them. These challenges get further magnified with the slowdown of the silicon CMOS (Dennard) scaling, as ICs incorporate tremendous design complexity (multiple processor

Digital Object Identifier 10.1109/MDAT.2016.2590987
Date of publication: 13 July 2016; date of current version: 20 October 2016.

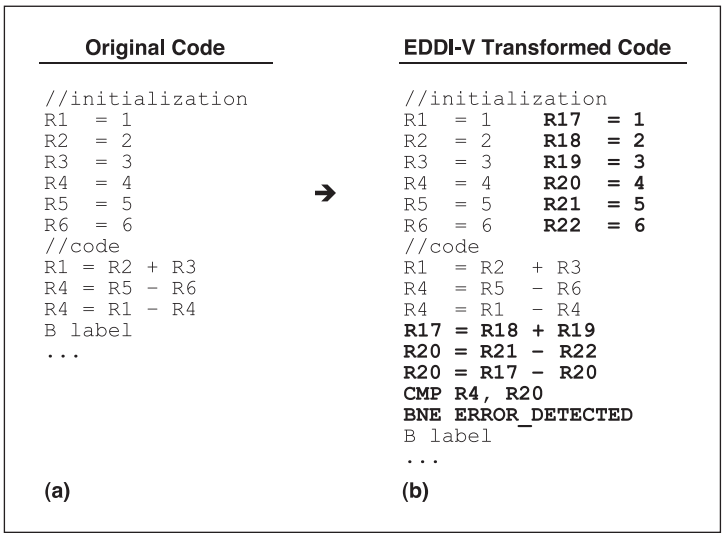

Figure 1. EDDI-V transformation with Inst_min = Inst_max = 3.

cores; accelerators; uncore components[1]; power, thermal, and reliability management) to meet performance and energy-efficiency targets.

A variety of tests (random instruction tests, architecture-specific focused tests, instruction traces, and end-user applications) are run on manufactured ICs to detect bugs. At-speed tests run on silicon are orders of magnitude faster than the simulations used in pre-silicon verification. Thus, a large number of tests can be run. However, coverage metrics for post-silicon validation tests are an open challenge. For manufacturing tests, various well-established test coverage metrics (e.g., single-stuck-at coverage) exist. These metrics have been experimentally shown to be effective in detecting manufacturing defects. Metrics such as code coverage and assertion coverage are used during pre-silicon verification, but are less standardized. For post-silicon validation, coverage metrics are in their infancy and are highly challenging (partially due to very limited controllability and observability with traditional post-silicon validation tests).

Simply detecting a bug during post-silicon validation is also not sufficient. Upon bug detection, post-silicon bug localization aims to identify a bug trace (a sequence of inputs, e.g., instructions, that activate and detect the bug) and the hardware design block containing the bug. Existing post-silicon validation and debug practices are mostly *ad hoc*, resulting in very high costs. The effort to localize bugs from observed system failures (e.g., deadlocks, crashes, output errors) often dominates overall cost. It might even take months of manual work to localize and debug a single bug [1], [2].

Question 3: Why are existing post-silicon validation and debug techniques insufficient?

Many existing techniques rely on trace buffers, small memories that record a selected set of signals, typically for 1000 clock cycles. Simply recording so few cycles of history is not sufficient, as explained in Question 4.

Failure reproduction (reexecuting the failure-causing stimuli from an error-free state) is also very difficult due to nondeterministic behaviors (interrupts, I/O functionalities, interactions between multiple processor cores, and operating system functionalities such as context switches).

The sheer size of current designs poses major challenges as well. System-level simulation is several orders of magnitude slower than actual silicon. The application of existing formal analysis and Boolean satisfiability for bug localization is severely limited by design size.

While assertions can be used for post-silicon validation and debug, manual assertion creation is difficult. Creating assertions that can be efficiently implemented in hardware is even more challenging. The number of assertions often explodes with automatic assertion generation, and many of these assertions are ineffective at catching bugs.

Question 4: What can we do to "really" advance post-silicon validation and debug?

We must first understand the "real" problem: very long error detection latencies, as demonstrated in [3]–[6]. Error detection latency is the time elapsed between when a test activates a bug, and when the generated error causes an observable failure (e.g., system crash, timeout, deadlock, exception). Error detection latencies for "difficult" bugs can exceed millions or even billions of clock cycles [4], [5]. It is extremely difficult to trace that far back in a system's operation, especially for today's complex SoCs, quickly making existing techniques ineffective.

[1]Uncore components refer to components in an IC that are neither processor cores nor coprocessors. Examples include interconnect fabrics and cache/memory controllers.

Question 5: How do you overcome long error detection latencies?

Our previous work, Quick Error Detection (QED), is highly effective at quickly detecting logic and electrical bugs inside processor cores, uncore components, accelerators, and components related to power-management features [3]–[7]. QED drastically reduces error detection latencies by several orders of magnitude. Since bugs are detected much quicker, QED simultaneously improves bug coverage.

QED consists of a set of transformations that systematically convert a wide variety of existing tests (referred to as original tests) into new QED family tests [3], [4]. QED transformations can also be implemented entirely in software, and are consequently readily applicable to existing ICs. Augmenting QED with hardware (e.g., [6] and [7]) can further improve error detection latency, coverage, and test execution time.

The software-only QED transformations are Error Detection Using Duplicated Instructions For Validation (EDDI-V), Control Flow Checking Using Software Signatures For Validation (CFCSS-V), Control Flow Tracking Using Software Signatures For Validation (CFTSS-V), and Proactive Load and Check (PLC) as detailed in [3] and [4]. We provide examples of EDDI-V and PLC below.

EDDI-V targets bugs inside processor cores by frequently checking the results of original instructions against the results of duplicated instructions created by EDDI-V. First, the register and memory spaces are divided into two halves,[2] one each for the original instructions and the duplicated instructions. For every load, store, arithmetic, logical, shift, or move instruction in the original test, EDDI-V creates a duplicate instruction that performs the same operation, using the duplicate registers and memory. Both instruction streams execute in the same order, but are intertwined. To compare intermediate results from both instruction streams, the transformation inserts frequent check instructions of the following form:

```
CMP Ra, Ra'
```

where *Ra* and *Ra′* are the original and (corresponding) duplicate registers, respectively. A mismatch in any check instruction indicates an error (i.e., the QED test fails).

[2]For EDDI-V, if it is not possible to divide the registers into two halves (i.e., if the original test needs to use all of the available registers), we can use memory to store the register values [5].

Transformed Code	PLC Operation

...
<PLC Operation>
R1 = R2 + R3
R4 = R5 – R6
R17 = R18 + R19
R20 = R21 – R22
<PLC Operation>
R7 = R1 – R4
R9 = R7 * R8
R23 = R17 – R20
R25 = R23 * R24
<PLC Operation>
...
	for <A,A'> in PLC_list do
	LOCK(A)
	LOCK(A')
	Rt = LOAD(A)
	Rt' = LOAD(A')
	UNLOCK(A')
	UNLOCK(A')
	CMP Rt, Rt'
	BNE ERROR_DETECTED
	end for

(a)	(b)

Figure 2. PLC example with Inst_min = Inst_max = 4.

QED implementations can be intrusive; i.e., a transformation might prevent a bug from being detected. To address intrusiveness [5], the insertion of the duplicated and check instructions can be controlled by the parameters Inst_min and Inst_max, the minimum (maximum) number of instructions from the original test that execute before any duplicated or check instructions execute.

PLC targets bugs inside uncore components by proactively performing frequent loads from memory (through those uncore components) and checking the values loaded. Starting with an EDDI-V-transformed QED test, PLC inserts Proactive Load and Check operations throughout the test, which runs on all cores and all threads. Figure 2a shows code transformed with PLC operations, which are detailed in Figure 2b. Each PLC operation checks the values in memory for a selected set of variables (PLC list). For each selected variable, a PLC operation compares the value from the memory reserved for original instructions (address *A*) and duplicated instructions (address *A′*). Any mismatch during the PLC check (*CMP Rt, Rt'*) indicates an error. Several PLC strategies are discussed in [5] and [6].

We demonstrated the effectiveness of QED on multiple hardware platforms (e.g., Intel Core i7 SoC) using "difficult" bug scenarios from commercial multicore SoCs [3]–[7]. QED shortens error detection latencies by up to nine orders of magnitude (e.g., from billions of clock cycles to around ten cycles for a multicore Freescale SoC). Furthermore, QED enables up to a fourfold increase in bug coverage.

Question 6: What is the difference between QED and symbolic QED?

Symbolic QED [8] is based on QED principles. It can be used for both pre-silicon verification and

post-silicon validation. It uses Bounded Model Checking (BMC) [9] for logic bug detection and localization by analyzing the RTL design. Given a system model and a property to be checked, BMC formally analyzes the model to find whether the property can be violated within a bounded number of steps. If it can be violated, then BMC reports a counterexample showing the steps required.

Symbolic QED uses the design RTL for the system model. For the property, it checks whether a QED test exists that could fail (detailed in Question 8). The BMC tool searches the space of all possible QED tests (within its bound). In [8], symbolic QED is based on the EDDI-V and PLC transformations; however, it can also be expanded for CFCSS-V and CFTSS-V.

Symbolic QED localizes bugs in two ways. If the BMC returns a failing QED test, then it is not possible to find a shorter failing QED test (with the corresponding QED transformations) that activates the same logic bug [9]. Also, symbolic QED uses partial instantiations to help localize the bug to various parts of the design (Question 10).

Question 7: What are the key benefits of symbolic QED?

1) Symbolic QED is applicable to any SoC design as long as it contains at least one programmable processor core (a generally valid assumption).
2) It is broadly applicable for logic bugs inside processor cores, accelerators, and uncore components.
3) It does not require failure reproduction or simulation.
4) It is a fully automated approach for logic bug localization without requiring human intervention.
5) It does not require any additional hardware to localize logic bugs, a significant advantage over other techniques that must add area overhead to the design (e.g., for trace buffers).
6) It does not require design-specific assertions and provides a very succinct and generic property to quickly detect and localize logic bugs.

Question 8: How do you create formal properties for symbolic QED?

Creating "good" properties to check using formal techniques (such as BMC) is a known difficult challenge. Symbolic QED fully automates this process by using a universal property (i.e., a property that is effective for finding a large class of bugs, regardless of the specific hardware blocks present in the SoC) that is based on QED testing. The property provided to BMC is derived from the check that would detect the error during a QED test. The BMC tool attempts to find a counterexample to the following property:

$$\wedge_{\alpha \epsilon \{0..\frac{n}{2}-1\}} Ra == Ra'$$

where n is the number of registers defined by the ISA. Here, (for a $\in \{0..n/2-1\}$), Ra and Ra' correspond to registers allocated for original instructions and duplicated instructions, respectively. This property identifies bugs detected both by EDDI-V and PLC QED tests.

Question 9: How do you create environmental constraints for symbolic QED?

If a BMC tool does not impose any constraints on its inputs while searching for a counterexample, it may find counterexamples that consist of invalid inputs. Consequently, environmental constraints (constraints that disallow inputs that would not be seen in actual deployment) must be added, and this provides another challenge for formal analysis.

In symbolic QED, adding environmental constraints is straightforward. Specifically, the BMC tool must only consider sequences of instructions that correspond to QED tests. We accomplish this without the extensive manual work required in traditional formal analysis using two mechanisms.

1) We constrain the inputs to the instruction fetch unit (of each processor core) to be arbitrary but valid instructions, directly obtained from the Instruction Set Architecture (ISA).
2) We add a new QED module to the fetch unit of each processor core during BMC (details in [8]). The QED module automatically performs the EDDI-V transformation on-the-fly for any input sequence explored during BMC. The QED module is only used within the BMC tool and is not added to the manufactured IC; i.e., there is no performance/area/power overhead, even in post-silicon validation. Including load and store instructions enables our approach to activate and detect bugs inside uncore components as well

as processor cores. The QED module is quite simple and needs to be designed only once for a given ISA.

Figure 3 shows an example of transformations by the QED module. Figure 3a shows the sequence of input instructions selected by the BMC tool (BMC can select any sequence of valid instructions). Figure 3b shows the actual instructions executed by the processor [LOAD(A) is duplicated as LOAD(A')]. Thus, comparing the registers (using the BMC tool) is equivalent to a PLC check on variables A and A'. There are four events here: 1) store to A by core 1; 2) load from A by core 2; 3) store to A' by core 1; and 4) load from A' by core 2. As explained in [8], to avoid false fails without using locks, the QED module ensures that the order of 3) and 4) is the same as the order of 1) and 2), even if multiple cores load from A and A'.

```
(a)
Core 1                          Core 2
A  = STORE(R1)                  R2 = R3 - R4
R2 = R3 + R4                    R1 = LOAD(A)
R5 = LOAD(A)                    R5 = LOAD(B)
BRANCH label

(b)
Core 1                          Core 2
A  = STORE(R1)                  R2 = R3 - R4
R2 = R3 + R4                    R1 = LOAD(A)   // PLC load
R5 = LOAD(A)  // PLC load       R5 = LOAD(B)
A' = STORE(R17)                 R18 = R19 - R20
R18= R19 + R20                  R17 = LOAD(A') // PLC Load
R21= LOAD(A') // PLC load       R21 = LOAD(B')
BRANCH label
```

Figure 3. Example of QED transformation by the QED module: (a) the original instruction sequence on cores 1 and 2; and (b) the actual transformed instructions executed.

The starting architectural state of the BMC tool must also be QED consistent, with matching original and duplicate registers and memory locations. This can be a reset state but runtimes improve by up to 5× [8] if the initial state is obtained from a fast, high-level simulation of a benchmark stopped when all QED values are consistent.

Question 10: How can symbolic QED scale for large designs?

A major challenge with BMC is handling large designs, which can significantly slow down a BMC tool or even cause it to fail while loading the design. However, symbolic QED does not require analysis of an entire design at once. A key property of QED checks is that they are compositional, i.e., they are preserved across partial instances of a design (as long as the instance has at least one processor core).

Symbolic QED handles large designs through partial instantiation using two design reduction techniques (details in [8]). Technique 1 takes all components with multiple instances, and repeatedly reduces their count by half (until there is only one left). For example, in a multicore SoC, the processor cores are removed until there is only one processor core left. Technique 2 removes a module if its removal does not divide the design into two disconnected components. For example, if a design has a processor core connected to a cache through a crossbar, the crossbar is not removed (without also removing the cache). Combinations and repetitions of the two techniques are considered when producing partial instances (with at least one processor core) for BMC. The partial instances can be analyzed by the BMC tool independently (in parallel).

We successfully demonstrated this technique on a 500-million-transistor, multicore design in [8].

Question 11: What symbolic QED results have been published?

In [8], we demonstrated the effectiveness of symbolic QED on the OpenSPARC T2 SoC (http://www.opensparc.net), an open-source version of the UltraSPARC T2, a 500-million-transistor SoC. For BMC, we used Mentor Graphics' Questa Formal tool (version 10.2c_3). We simulated a wide variety of "difficult" logic bugs (related to processor cores, uncore components, and power management features) that occurred in various commercial multicore SoCs. They are considered difficult because they took a long time (days to weeks) to localize. We compared three approaches for obtaining bug activation traces: an original benchmark test (FFT from SPLASH-2 [10]), the QED transformed benchmark test, and symbolic QED with the BMC initialized to a QED-consistent state from the benchmark.

Table 1 shows clearly that symbolic QED automatically produces bug traces up to six orders of magnitude shorter than traditional post-silicon

Table 1 Results comparing original tests (no QED), QED tests, and symbolic QED with the (minimum, average, maximum) bug trace length in instructions and clock cycles and (minimum, average, maximum) BMC runtimes.

		Original (No QED)	QED	**Symbolic QED**
Processor core only	Bug trace length (instructions)	[643,551k,4.9M]	[324,57k,233k]	**[3, 3, 3]**
	Bug trace length (clock cycles)	[842,572k,5.1M]	[367,66k,265k]	**[13, 15, 16]**
	Coverage	50.0%	100%	**100%**
	BMC runtime (minutes)	N/A	N/A	**[22, 46, 90]**
	Bugs localized	0%	0%	**100%**
Uncore	Bug trace length (instructions)	[620,1.6M,9.8M]	[231,59k,232k]	**[3, 4, 4]**
	Bug trace length (clock cycles)	[722,1.9M,11M]	[292,72k,289k]	**[14, 22, 29]**
	Coverage	55.3%	100%	**100%**
	BMC runtime (minutes)	N/A	N/A	**[78,164,188]**
	Bugs localized	0%	0%	**100%**
Power management	Bug trace length (instructions)	[1.5k,236k,495k]	[10k,68k,302k]	**[5, 5, 5]**
	Bug trace length (clock cycles)	[1.9k,251k,512k]	[13k,75k,319k] [2k,49k,149k]	**[17, 19, 22]**
	Coverage	66.7%	100%	**100%**
	BMC runtime (minutes)	N/A	N/A	**[205,266,333]**
	Bugs localized	0%	0%	**100%**

validation test that rely on end-result checks, and up to five orders of magnitude shorter than the QED test. Figure 4 shows the number of components the bugs were localized to by symbolic QED. Each bug can be further localized from the specific activation trace obtained. Finally, symbolic QED correctly and automatically produces short bug traces for all bugs in less than 7 h, without trace buffers or any other additional hardware. Several strategies for further improving BMC runtimes for partial instances are discussed in [8].

Question 12: How is symbolic QED an automated approach?

Symbolic QED requires no (or minimal) human intervention. It does not require manual generation of formal properties or environmental constraints. (The QED module is designed once for each ISA, and automatically invoked.) The partial instances (Question 10) are also created automatically. Provided that the BMC tool is set up to run through each partial instantiation, no further input is required.

Question 13: Is symbolic QED limited to bugs inside processor cores?

No, as shown in Table 1 and Question 8 (further detailed in [8]). We simulated a variety of bug scenarios, abstracted from actual "difficult" bugs that occurred in commercial multicore SoCs. These included logic bugs inside uncore components (i.e., cache controllers, memory controllers, and on-chip interconnection networks), power-management-related bugs, and processor core bugs. Symbolic QED successfully localized all of these bug scenarios, simulated on the OpenSPARC T2 SoC.

Question 14: How can symbolic QED be used for post-silicon validation and debug?

Some logic bugs can be very difficult to activate, because they require a very specific state that can only be reached by executing many specific instructions. As a result, during pre-silicon verification, symbolic QED may fail to detect this type of bug if it is not started from a close enough initial state; the trace would be too long for the BMC tool to discover. During post-silicon validation, however, QED tests can detect these bugs. The corresponding architectural states (prior to bug detection) can then be used to initialize the BMC tool for symbolic QED, making it possible to generate short bug activation traces and localize bugs.

Symbolic QED can be further enhanced during post-silicon validation through a bit of additional hardware (1%–2%) to reduce the size of the design analyzed by the BMC tool. This hardware does require QED post-silicon validation tests to be run first. In [8], we introduced small hardware structures,

change detectors, to monitor signals between hardware blocks and record if changes occur within a sliding window of immediately preceding execution cycles (e.g. the last 1000 cycles). When a QED test detects an error, if no signal changes are observed at the boundary of an uncore component (during the execution window being analyzed), then we exclude that component as a candidate contributing to the bug. This reduces the number of components for the BMC tool to analyze. The short error detection latencies with QED (typically less than 1000 cycles) allow for a small monitored sliding window, reducing the area impact.

In bug localization experiments using the OpenSPARC T2 SoC, highly utilized components (caches, memory controllers, crossbar) usually could not be excluded by change detectors. However, peripheral components with sporadic activities during post-silicon validation (e.g., I/O modules) could frequently be removed. Monitoring the latter required an area overhead of just 0.98% (versus 1.86% for the full design).

Question 15: What are the limitations of symbolic QED?

Ensuring that the design size is small enough to be analyzed by the BMC tool will remain a key challenge of implementing symbolic QED. We have demonstrated on the OpenSPARC T2 how partial instantiation and change detectors (during post-silicon validation) can address this problem. We are actively seeking larger designs to further test these approaches. Choosing a good initial state is another challenge, especially when symbolic QED is used during pre-silicon verification (as discussed in Question 14, QED tests can help with this problem during post-silicon validation).

Question 16: What are some of the future directions for symbolic QED?

1) Electrical bugs: Extending symbolic QED to localize electrical bugs requires two additional features: a model of the digital impact of electrical bugs, and an efficient technique to record the state of the IC during post-silicon validation using QED. Formal analysis of the captured error trace could then be used to localize electrical bugs. The use of symbolic QED for performance bugs (design errors that might affect an IC's performance while preserving correct logic functionality) is an open research question.
2) Software verification and debug: As demonstrated in [7], hybrid QED (H-QED) quickly detects bugs in C/C++ programs corresponding to high-level design descriptions of hardware accelerators (in addition to electrical bugs in hardware implementations). H-QED improves error detection latencies by up to two orders of magnitude and bug detection coverage by 3×. Since C/C++-level bugs were successfully detected by H-QED, symbolic QED approaches can be potentially explored for general software verification.
3) Analog/mixed-signal blocks: Partial instantiation in symbolic QED (Question 10) can omit analog/mixed-signal blocks as needed. If a bug is not localized within the digital blocks, this is strong evidence that it originates from the analog/mixed-signal blocks. If digital models for analog/mixed-signal blocks exist (e.g., in [11]), symbolic QED can be used for those blocks as well.
4) Full system-level fault localization: Symbolic QED, together with QED, can potentially enable full system-level fault localization of large-scale systems consisting of several ICs. This is a highly challenging problem, as explained in [12]. Once QED tests identify incorrect ICs in the system, symbolic QED may be used to localize and root-cause faults inside those ICs.
5) Diagnosis of manufacturing defects during functional testing and system-level testing: Functional tests and system-level tests routinely supplement scan tests to detect manufacturing defects in ICs. However, diagnosing manufacturing defects from functional test/system-level test failures is extremely difficult. Fault diagnosis is

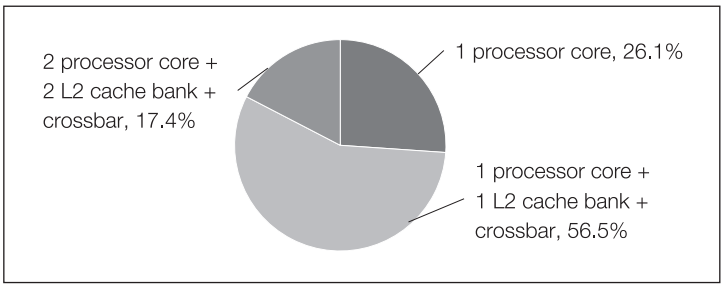

Figure 4. Percentage breakdown (by candidate modules) of bugs localized by symbolic QED.

important for several reasons including yield learning, test quality improvement, and improving IC reliability. A combination of QED and symbolic QED, together with fault models used for manufacturing testing, could enable effective fault diagnosis during functional and system-level testing.

References

[1] J. Keshava et al., "Post-silicon validation challenges: How EDA and academia can help," in *Proc. IEEE/ACM Design Autom. Conf.*, 2010, DOI: 10.1145/1837274.1837278.

[2] K. Reick, "Post-silicon debug—DAC workshop on post-silicon debug: Technologies, methodologies, and best-practices," in *Proc. IEEE/ACM Design Autom. Conf.*, 2012.

[3] T. Hong et al., "QED: Quick error detection tests for effective post-silicon validation," in *Proc. IEEE Int. Test Conf.*, 2010, DOI: 10.1109/TEST.2010.5699215.

[4] D. Lin et al., "Quick detection of difficult bugs for effective post-silicon validation," in *Proc. IEEE/ACM Design Autom. Conf.*, 2012, pp. 561–566.

[5] D. Lin et al., "Effective post-silicon validation of system-on-chips using quick error detection," *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 33, no. 10, pp. 1573–1590, 2014.

[6] D. Lin et al., "Quick error detection tests with fast runtimes for effective post-silicon validation and debug," in *Proc. IEEE Design Autom. Test Eur. Conf.*, 2015, DOI: 10.7873/DATE.2015.1012.

[7] D. Campbell et al., "Hybrid quick error detection (H-QED): Accelerator validation and debug using high-level synthesis principles," in *Proc. IEEE/ACM Design Autom. Conf.*, 2015, DOI: 10.1145/2744769.2744853.

[8] D. Lin, E. Singh, C. Barrett, and S. Mitra, "A structured approach to post-silicon validation and debug using Symbolic Quick Error Detection," in *Proc. IEEE Int. Test Conf.*, 2015, DOI: 10.1109/TEST.2015.7342397.

[9] E. Clarke et al., "Bounded model checking using satisfiability solving," *Formal Methods Syst. Design*, vol. 19, no. 1, pp. 7–34, 2001.

[10] S. C. Woo et al., "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. ACM/IEEE Int. Symp. Comput. Architect.*, 1995, pp. 24–36.

[11] A. V. Karthik et al., "ABCD-NL: Approximating continuous non-linear dynamical systems using purely Boolean models for analog/mixed-signal verification," in *Proc. Design Autom. Conf.*, 2014, pp. 250–255.

[12] Z. Conroy et al., "A practical perspective on reducing ASIC NTFs," in *Proc. IEEE Int. Test Conf.*, 2005, DOI: 10.1109/TEST.2005.1583992.

Eshan Singh is currently working toward a PhD at Stanford University, Stanford, CA, USA. His research interests include verification, post-silicon validation, VLSI design, 3-D ICs, and computer architecture. Singh has an MS in electrical engineering from Stanford University.

David Lin has a PhD in electrical engineering from Stanford University, Stanford, CA. His research interests include post-silicon validation, verification, debug, and computer architecture.

Clark Barrett is an Associate Professor (Research) of Computer Science at Stanford University, Stanford, CA, USA, with expertise in constraint solving and verification. Before joining Stanford, he was an Associate Professor at New York University, New York, NY, USA. His work has been recognized with a DAC best paper award, an IBM Software Quality Innovation award, the Haifa Verification Conference award, and top honors at the SMT, CASC, and SyGuS competitions.

Subhasish Mitra directs the Robust Systems Group in the Department of Electrical Engineering and the Department of Computer Science of Stanford University, Stanford, CA, USA, where he is the Chambers Faculty Scholar of Engineering. Before joining Stanford, he was a Principal Engineer at Intel. His research interests include robust systems, VLSI design, CAD, validation and test, nanosystems, and emerging neuroscience applications. He is a Fellow of the ACM and the IEEE.

■ Direct questions and comments about this article to Eshan Singh, Department of Electrical Engineering, Stanford University, Stanford, CA 94305 USA; esingh@stanford.edu.