Challenges and Trends in Modern SoC Design Verification

Wen Chen, Sandip Ray, and Jayanta Bhadra
NXP Semiconductors

Magdy Abadir
Helic Inc.

Li-C Wang
University of California at Santa Barbara

Editor's note:
This paper provides a tutorial overview of the state-of-the-art in verification of complex and heterogeneous Systems-on-Chip. The authors discuss current industrial trends and key research challenges.
—Haralampos Stratigopoulos, Sorbonne Universités, UPMC, CNRS, LIP6

■ **HARDWARE VERIFICATION TODAY** is a relatively mature topic, both in research and in industrial practice. Verification research dates back to at least three decades, with a rich body of literature [14], [21]. In industrial practice, verification is now firmly established as an integral component of the system development flow. Unfortunately, in spite of these advancements, there remains a significant gap between the state of the art in the technology today and the verification needs for modern industrial designs. The situation is exacerbated by the rapidly changing design ecosystem as we move rapidly and inevitably to the era of automated vehicles, smart cities, and Internet of Things (IoT). In particular, this new era has ushered in an environment where an electronic device first collects, analyzes, and stores some of our most intimate personal information, such as location, health, fitness, and sleep patterns; then communicates such information through a network of billions of other computing devices; and finally operates without pause or halt even when that environment may include millions of potentially malicious or otherwise compromised devices. As system design and architecture get transformed to adapt themselves to this new ecosystem, verification must adjust as well [31].

A critical impact (but not the only one) on verification in the new era is the resources available. With the demand to churn out billions of diverse computing devices, time-to-market requirements for design and system development have become more aggressive than ever before. For example, a typical microprocessor life cycle from exploration to start of production used to range between three and four years; for some IoT devices, this has shrunk to less than a year. Such aggressive shrinkage obviously implies inadequate time for thorough design review, potential misunderstanding of specification and requirements from various developers and stake-holders on functional decomposition of the design, and a consequent increase in errors. On the other hand, the shrinking life cycle also means less time for verification. Consequently, the demand from verification has been to handle potentially more error-prone designs than before, with even less time and fewer resources. One consequence of this aggressive scheduling has been more in-field escapes and

Digital Object Identifier 10.1109/MDAT.2017.2735383
Date of publication: 3 August 2017; date of current version: 13 September 2017.

requirements for in-field patching of devices and systems, potentially through software and firmware updates. Another, perhaps positive, consequence has been a trend toward focused development of verification methodologies for achievable and high-value targets, e.g., security, networking, and cyber-physical components.

In this paper, we discuss several challenges in SoC design verification in this new era. Some of the challenges are classical problems in verification, e.g., tool scalability, reuse of verification collateral across systems and designs, and so on. In addition, we discuss some of the newer challenges ushered in specifically by the connected ecosystem of IoT. Finally, we discuss emerging trends in industrial verification tools and methodologies to address some of these challenges.

Verification life cycle

Most electronic devices today are architected through an SoC design paradigm: the idea is to develop a system through integration of predesigned hardware and software blocks, often collectively referred to as design intellectual properties (IPs). In current industrial setting, IPs are typically developed independently, either in-house or by third-party vendors. An SoC integration team collects and assimilates these IPs based on the system requirement for the target device. To enable smooth integration of the IPs into the target system, they are designed to communicate with each other through well-defined interfaces, e.g., ARM provides the AMBA bus interface that includes on-chip interconnect specification for the connection and management of various functional blocks. In the context of SoC designs, verification involves two somewhat independent verification flows, one for ensuring correct operation of the IPs (and their adherence with the interface protocols) and another for the assembled system.

Given the complexity of modern computing devices, both IP and SoC verification flows today are significantly complex, requiring careful upfront planning, and span almost the entirety of the design life cycle. In this section, we give an overview of the various components of verification in current industrial practice, as shown in Figure 1. Obviously, the notion of “industrial practice” is somewhat of a misnomer, since it varies from company to company based on business targets, product needs, and even legacy practices. Nevertheless, the following description captures the basic essence of the SoC design verification flow and is relatively general.

Verification planning

This activity starts about the same time as the product planning, and continues through the system development phase. Product planning requires definition of the various IPs necessary, their decomposition into hardware and software components, the connection and communication interfaces, and various power, performance, security, and energy targets. Correspondingly, verification planning includes creation of appropriate test plans, test cards, definition of specialized design blocks called verification IPs (VIPs) instrumentation in the design for post-silicon debug, definition of various monitors, checker, exercisers, and so on.

Architecture verification and prototype definitions

One of the first stages in the definition of an SoC design is the system architecture, which defines various functional parameters of the design, communication protocols among IPs, power and performance management schemes, and so on. The parameters and design features explored at this stage include cache size, pipeline depth, protocol definitions for power management and security, and so on. The exploration is performed through a variety of “architectural models,” which simulate typical workloads and target use cases of the device, and identify parameter values that satisfy the device targets (e.g., power, performance, and security) identified in the planning stage. There are two important verification activities during this architectural exploration stage. The first is the functional verification of the various communication protocols. This activity allows detection of high-level protocol errors at an early stage when the design models are abstract and consequently simple, and the design is relatively less mature; such errors, if they escape into the product implementation, can become extremely expensive, since a fix at that stage might require major redesign of multiple IPs. Given the high abstraction of design models at this stage, it is feasible to perform formal analysis to achieve this [35]; in current practice, formal methods are augmented with high-level simulation to provide the desired coverage. The second crucial role for verification is to initiate the development of hardware prototyping models for subsequent needs in software and firmware verification. To understand this need, note that low-level software and firmware programs need to be validated for correctness when operating on the target (and evolving) hardware design developed during the implementation phase (see below). Clearly, one cannot wait for the hardware implementation to be stabilized before initiating software/firmware verification. Consequently, high-level software models of the hardware, also referred to as virtual prototype models, are developed to enable accelerated software/firmware verification. These models are typically at the same abstraction level as the architecture models (and sometimes derived from the latter), but they are different and serve a different purpose. Unlike architectural models, prototype models are written to provide a hardware abstraction that nevertheless exercises various software corner cases. One key requirement from the above is that the prototype model must include definition (and abstract functionality) of all the software-visible interface registers of the various IPs. Development of prototype models is initiated concurrently with architectural models, and it continues into the RTL development time-frame. The models are usually coordinated with various "drops" or releases, each containing functionality at various degrees of maturity; these drops are coordinated and synchronized carefully within the time-frame of software validation targets.

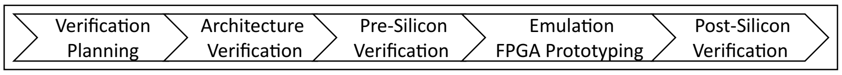

Verification Planning
Architecture Verification
Pre-Silicon Verification
Emulation FPGA Prototyping
Post-Silicon Verification

Figure 1. Verification life cycle.

Pre-silicon verification

This is the major resource-intense verification activity that takes place during (and after) hardware development and implementation. Note that this is a continuous process, with increasing level of maturity and complexity as the design matures. Most industrial SoC designs include a combination of legacy and new IPs, some created in-house and some collected from third-party IP providers. An IP verification team (whether in-house or third-party) performs the verification of the IP being delivered. This is done in a standalone environment, i.e., the objective is to ensure that the IP on its own functions as expected. Subsequently, the SoC team would integrate the IPs into an (evolving) SoC model and perform system-level verification; the target of the system-level verification is to ensure that the IPs function correctly together as an integrated system. An IP is delivered to the SoC integration team either as a hard IP, i.e., formatted as a physical design layout, or as a soft IP, in the form of an RTL or design netlist. The amount of verification performed by the IP team depends on the form in which the IP is delivered (e.g., a hard IP includes significantly higher verification requirement than a soft IP). Traditionally, IP verification has entailed exercising (and ensuring correctness of) the IP design in a standalone environment. This permits a company to have a robust portfolio of generic IP designs that can be quickly integrated into various SoC design products. With this view, an IP verification team develops such a standalone verification infrastructure for the target IP. For simulation, this infrastructure includes testbench and environment definitions that capture the target use cases of the IP design; for formal verification, it may include environmental assumptions, target assertions, etc. More recently, there has been a strong push to avoid "over-validation", i.e., to validate an SoC design for only its target use cases (see below). This has an impact on IP validation, e.g., one has to define the use cases for the IP corresponding to the SoC use cases. When such (verified) IPs are delivered to the SoC integration verification team, they can then target system-level scenarios. Note that each use case requires communication among multiple IPs; this is why it is so important in planning to carefully define IP drops to enable cohesive system-level SoC verification. Most SoC integration verification includes system-level simulation and definition of various use cases. However, note that many use cases require co-execution of hardware and software modules. These are obviously difficult to exercise in simulation, since running software on RTL modules is slow and often infeasible; such use cases are generally deferred until the design is mature for emulation and FPGA prototyping (see below).

Emulation and FPGA prototyping

Technically, verification using emulation and FPGA prototyping is simply a part of pre-silicon verification, since they are performed before the system goes into fabrication. However, in practice, they form an important bridge between pre-silicon and post-silicon verification. Here one maps the RTL model of the hardware into a reconfigurable architecture such as FPGA, or specialized accelerators and emulators [5], [8], [10]; these platforms run about hundreds to thousands times faster than an RTL simulator; consequently, one can execute hardware/software use case scenarios such as an operating system boot in a few hours. This speed is obtained at the cost of controllability and observability. In a simulator, one can observe any internal signal of the design at any time. In contrast, in FPGA prototyping (which is the fastest of the pre-silicon platforms) the observability is restricted to a few thousands of internal signals. Furthermore, one must decide on the signals to be observed before generating the FPGA bit-stream. Reconfiguring the observability would require recompilation of the bit-stream which might take several hours. Consequently, they are used only when the design is quite mature, e.g., when the functionality is relatively stable and debug observability fixed enough to warrant few recompilations. Recent innovations within FPGA technology [2], [3] address some of the observability limitations in FPGA solutions. Nevertheless, observability and recompilation cost remain a challenge.

Post-silicon verification

Post-silicon validation is the activity where one uses an actual silicon artifact instead of an RTL model. To enable post-silicon validation, early silicon is typically brought into a debug lab, where various tests are run to validate functionality, timing, power, performance, electrical characteristics, physical stress effects, and so on. It is the last validation gate, which must be passed before mass production can be initiated. Post-silicon validation is a highly complex activity, with its own significant planning, exploration, and execution methodologies. A fuller discussion of post-silicon validation, as well as the specific challenges therein, is out of scope for this paper, and the reader can refer to a previous paper [29] for a complete discussion. From a functional perspective, the fact that a test can run at a target clock speed enables execution of long use cases (e.g., booting an operating system within seconds, exercising various power management and security features). On the other hand, it is considerably more complex to control or observe the execution of silicon than that of an RTL simulation model (or even FPGA or emulation models). Furthermore, changing observability in silicon is obviously infeasible.

Verification challenges: Traditional and emerging

In spite of maturity, verification tools today do not scale up to the needs of modern SoC verification problems. In this section, we discuss some of the key challenges. While some of the challenges are driven by complexity (e.g., tool scalability, particularly for formal), some are driven by the needs of the rapidly changing technology trends.

Shrinking verification time

The exponential growth in devices engendered by the IoT regime has resulted in a shrinkage in the system development life cycle, leaving little time for customized verification efforts. However, each device has a different use case requirement, with associated functionality, performance, energy, and security constraints. We are consequently faced with the conundrum of requiring to create standardized, reusable verification flows and methodologies that can be easily adapted to a diversity of electronic devices each with its unique tailor-made constraints. Two orthogonal approaches have so far been taken to address this problem. The first is to improve tool scalability with the goal of eventually turning verification into a turn-key solution; achieving this goal, however, remains elusive (see below). The other approach entails making the systems themselves highly configurable, so that the same design may be “patched” to perform various use cases either through software or firmware update or through hardware reconfiguration. Unfortunately, developing such configurable designs also has a downside. Aside of the fact that it is impossible to determine all the different use cases of a hardware system in advance (and hence identify whether enough configurability has been built in), this approach also significantly blows up the number of states of the system and consequently makes their verification more challenging.

Limited tool scalability

Scalability remains a crucial problem in effective application of verification technology. The problem is felt particularly acutely in formal verification; in spite of significant recent advances in automated formal technologies such as satisfiability (SAT) checking and SAT modulo theories (SMT) [12], the chasm between the scale and complexity of modern SoC designs and those which can be handled by formal technology has continued to grow. The increasing requirements for configurability and consequent increase in design complexity have only served to exacerbate the situation. To address this problem, there has been a growing trend in formal methods to target specific applications (e.g., security, deadlock, etc.) rather than a complete proof of functional correctness. We will discuss some of these applications in the following section.

The cost of simulation-based verification is also getting increasingly prohibitive as the design size continues to increase. For instance, random simulation at the SoC level can cover only a tiny portion of the design space. On the other hand, directed tests designed for specific coverage goals can be prohibitive in terms of human effort required.

Specification capture

A key challenge in the applicability of verification today is the lack of specifications. Traditionally, specifications have largely relied on requirements documents, which under-specified or omitted design behavior for some scenarios or left some cases vague and ambiguous. Such omissions and ambiguity, while sometimes intentional, were often due to the ambiguity inherent in natural languages. Unfortunately, the problem becomes significantly more contentious in the context of modern SoC designs than for traditional microprocessors. Recall that at least in the realm of microprocessors, there is a natural abstraction of the hardware defined by the instruction-set architecture (ISA). Although the semantics of ISA are complex (and typically described in ambiguous English manuals spanning thousands of pages), the very fact of their standardization and stability across product generations enables concretization and general understanding of their intended behavior. For example, most microprocessor development companies have a detailed simulator for the microprocessor ISA, which can serve as an executable golden reference. On the other hand, it is much harder to characterize the intended behavior of an SoC design. Indeed, SoC design requirements span across multiple documents (often contradictory) that consider the intended behavior from a variety of directions, e.g., there are system-level requirements documents, integration documents, high-level-architecture documents, microarchitecture documents, as well as cross-cutting documents for system-level power management, security, and post-silicon validation [30]. Merely reconciling the descriptions from the different documents is a highly complex activity, let alone defining properties and assertions as necessary for verification.

Use case identification

Given the aggressive time-to-market requirements, there has been a general move in verification today away from comprehensive coverage of the whole system (or a system component) and toward more narrowly defined coverage of intended usage scenarios. For example, for a device intended primarily for low-power and low-performance applications (e.g., a small wearable device), the intended usage would include scenarios where different components transition frequently into various sleep modes but would not include sustained execution at high clock speeds; conversely, a high-performance device such as a gaming system would prioritize execution at high clock speeds. In general, the exploration and planning phases of the device life-cycle define a set of use cases which constitute the target usages of the device and must be exercised during verification. Unfortunately, this approach, while attempting to reduce verification effort by eliminating “over-validation” might induce significant complexity in the process. In particular, the usage scenarios are typically defined at a level of the device and involve complex interaction of hardware, firmware, and software; it is nontrivial to determine from such high-level verification targets how to define verification goals for individual IPs, or even hardware blocks for the entire SoC. Furthermore, the SoC design itself and individual IPs have orthogonal verification needs, together with their own methodologies, flows, and timelines. For example, an USB controller IP is targeted to be developed (and verified) to be usable across the slew of USB devices; a smartphone making use of this IP, on the other hand, must be verified for the usage scenarios which are relevant for the smartphone. Finally, exercising the device-level use cases requires hardware, firmware, and software at a reasonable maturity, which is available only late in the system life cycle (e.g., either at post-silicon or at least during emulation of FPGA prototyping). Bugs found this late may be expensive to fix and may involve considerable design churn.

Power management challenges

Low power requirements for integrated circuits and power efficiency have been a main focus for today's complex SoC designs. Power gating and clock gating have been the most effective and widely used approaches for power reduction. Power gating relies on shutting off the blocks or transistors that are not used. Clock gating shuts off blocks or registers that are not required to be active. Industrial standards have been developed to describe the power intent of low power designs to support the simulation of power aspects at RTL simulation. However, these features significantly complicate verification activities. One reason is the obvious multiplication of complexity. It is not uncommon that a low power design can feature tens of power domains and thus hundreds of power modes. It is prohibitive to verify (whether through simulation or through formal methods) that the design is functional under all possible power modes. In practice, verification focuses on SoC use case scenarios, which are driven by hypervisor/OS control and application-level power management. This requires hardware/software co-verification of the power management features. A second—perhaps more subtle—challenge involves its interaction with post-silicon verification. The behavior within a power-gated IP cannot be observed during silicon execution; this implies that it is very difficult to validate design behaviors as various IPs get in and out of different sleep states. Unfortunately, these are exact states that account for subtle corner-case bugs, making validation challenging. To make matters worse, power-gated IPs may make it difficult to observe behavior of other IPs that are not in sleep states. Consider an IP A with an observable signal s. In order for s to be observable, its value must be routed to an observation point such as an output pin or system memory. If this route includes another IP B then we may not be able to observe the value of s whenever B is power-gated even if A is active at that time.

Security and functional safety

Security and privacy have become critical requirements for electronic devices in the modern era. Unfortunately, these are often poorly specified, and even poorly understood. One reason is that with the new IoT era, devices are getting connected which were never originally intended to be connected, e.g., refrigerators, light bulbs, or even automobiles. Consequently, security threats and mitigation remain unclear and one typically resorts to experts performing "hackathons," i.e., directed targeted hacking of the device, to identify security threats. In addition to security, functional safety, i.e., the assurance that the device does not harm anything in the environment due to system failure is a critical requirement for electronic devices used in applications such as aerospace and automotive. Safety mechanisms must be implemented for such devices to ensure that the device can be functional under the circumstances of unexpected errors. For example, lockstep systems are fault-tolerant systems commonly used in automotive devices that run safety critical operations in parallel. It allows error detection and error correction: the output from lockstep operations can be compared to determine if there has been a fault if there are at least two systems (dual modular redundancy), and the error can be automatically corrected if there are at least three systems (triple modular redundancy), via majority vote. Safety critical devices must be compliant with IEC 61508 [13], and ISO 26262 [23] is particularly designed for automotive electronics.

Hardware/software co-verification

In the days of microprocessors and application software, it was easy to separate concerns between hardware and software verification activities. However, today, with an increasing trend of defining critical functionality in software, it is difficult to make the distinction. Indeed, it may not be possible in many cases to define a coherent specification (or intended behavior) of the hardware without the associated firmware or software running. This has several consequences for verification. In particular, hardware and software are traditionally developed independently; the tight coupling of the two makes it incumbent that we define software milestones to closely correspond to (and be consistent with) various RTL drops. Furthermore, validating software requires an underlying hardware model that is stable, mature, and fast.

An RTL model and simulation environment does not have any of these characteristics. On the other hand, waiting for the maturity of emulation or silicon may be too late for identifying critical errors.

Analog and mixed signal components

Most IoT devices include various sensors and actuators in addition to their digital processing core, as the environment in which these devices operate is inherently analog. As a result, an increasingly large portion of the die area is occupied by analog/mixed-signal (AMS) circuits [27]. Due to the complex nature of analog behavior, design and verification methodologies of analog circuits are far more primitive compared with that of digital circuits. Verification of AMS ICs remains complex, expensive, and often a "one-off" task. Complicating the problem is the requirement of combining both methodologies to ensure thorough verification of comprehensive aspects of the mixed-signal SoCs.

Verification trends

Over the last few years, Foster led several industry studies to identify broad trends in verification [20]. One can make the following critical observations from these studies:

- Verification represents bulk of the effort in the system design, incurring on average a cost of about 57% of the total project time. There has been a discernible increase in the number of projects where verification incurred the cost of over 80% of the project time.
- Most designs show an increase in the use of emulation and FPGA models, with more usage of these technologies the more complex the design. This is consistent with the need for a fast prototyping environment, particularly for complex SoCs, and also perhaps emphasizes the role of software in modern systems (which require emulation/FPGA prototyping for validation).
- Most successful designs are productized after an average of two silicon spins. Note that for a hardware/software system this translates to one spin for catching all hardware problems and another for all the software interaction issues. This underlines the critical role of *pre-silicon* verification to ensure that there are no critical gating issues during post-silicon validation.There has been a significant increase in the use of both simulation-based verification and targeted formal verification activities.

The last point above deserves qualification. In particular, recall that both simulation and formal verification techniques are falling short of the scalability requirements of modern computing devices. How then are they being increasingly adopted?

The answer lies in transformative changes that have been occurring in these technologies in the recent years. Rather than focusing on full functional coverage, they are being targeted toward critical design features. Current research trends include verification of specific emerging application domains, such as automotive and security, and using data analytics to improve verification efficiency. In the remainder of this section, we dive a little deeper into how these technologies have been changing to adapt themselves to the increasing demand as well as to address the scalability gap.

Simulation technologies

Simulation is the mainstay for verifying complex SoCs thanks to its scalability to large industrial designs. State-of-the-art simulation-based verification methodologies include a highly automated process that includes test generation, checking, and coverage collection, combined with islands of manual labor [41]. In the beginning, a verification plan is defined to capture all the features required to be verified. Then stimuli (tests) are either manually crafted by verification engineers or automatically generated by a test generator. The stimuli are applied to the simulation environment and the behavior of the design is monitored and checked against expectation. To measure the completeness of verification, coverage metrics are defined in terms of which area of the design and which design functionality are exercised in simulation. Metric (coverage) driven verification has been adopted as an industrial paradigm where coverage is used to monitor and control the verification process. A sufficient coverage level is required for verification sign-off.

Simulation and test generation are the two core technologies in simulation-based verification that have been continuously pushed forward over the years. We highlight the recent technological trends and advances of simulation technologies in this subsection and that of test generation in the following one.

Simulator is the backbone of simulation-based verification environment and all major EDA vendors have their offerings.

The technology of digital simulators is a highly mature area where the speedup of simulation on a single machine is incremental and heavily relies on the performance boost of the underlying machine brought by technology scaling. As it gets more challenging to improve performance by technology scaling, there have been endeavors to leverage parallel computation to accelerate simulation. However, there have been challenges to parallelize simulation due to the event-driven nature of simulators and the complex dependencies between the RTL code blocks and expressions. Common simulators process RTL code in a single thread, managing a single active queue of events and handling them one at a time. This serialized processing fashion is inherently difficult to parallelize and accelerate by adding more computation power. In order to parallelize the simulation, a desired solution needs to break the dependencies between the RTL code blocks and partition them into semi-independent elements, each of which can be run in a thread with minimal synchronization between threads.

In recent years, more advances in the simulator technology have emerged in the capabilities to support simulation of aspects that were not considered ones of conventional digital simulation. The past few years have witnessed the impact on the SoC designs from emerging application areas such as IoT and automotive. Accordingly, various design features have been adopted in SoC designs to accommodate the requirements of these applications. Following are some illustrative examples of verification needs arising from such features.

Analog and mixed signal verification

Depending on the portion of AMS circuits on chip, the testbench architecture for AMS simulation can be divided into two categories: "analog on top" (top-level models are analog with digital modules inside) or "digital on top" (top- level models are digital with analog modules inside). The latter is more commonly used and the proper modeling of analog behavior is critical to "digital on top" mixed signal chip verification. Analog models of different abstraction levels are used through the project life cycle, with consideration of the tradeoff between simulation speed and accuracy. For example, Verilog-AMS provides four abstraction levels to model analog behaviors [9]. To support AMS verification, the simulator must have the performance and capacity to simulate a mixture of models at different abstraction levels for today's increasingly large designs in a reasonable amount of time, while maintaining an acceptable level of accuracy. It is not uncommon that there are nested digital and analog blocks in complex SoC designs, which should also be supported by the simulator. In addition, the co-existence of models at various abstraction levels creates complexity in verification planning as the models can be mixed and matched for achieving different verification goals.

Short of the capability of automatic abstract modeling of the behavior of low-level models, today's AMS simulation strongly relies on the user to provide the abstraction models. Proper modeling of analog behavior requires in-depth knowledge and know-how, and the modeling efforts can be comparable to design efforts. Due to the continuous nature of analog behavior, the checkers in AMS simulation largely differ from their digital counterparts and are difficult to implement correctly. For example, a voltage overshoot might be missed by a checker if signal sampling is not setup appropriately. In addition, not only time-domain properties but also frequency-domain properties need to be checked for analog circuits. It is highly desirable that advances of AMS simulation technology can address or at least mitigate these challenges.

Power-aware verification

While clock gating is usually implemented in RTL code, power gating is implemented by capturing the power intent in a standard format and instrumenting the necessary circuitry during synthesis [1], [7]. The power intent file is also used in simulation with RTL code to find out design flaws of the low-power SoCs at an early design stage. Power-aware simulators generally support simulating the behavior resulted from the power gating circuitry specified in power intent files, such as power switches, isolation cells, and retention registers.

Besides power gating, the requirements of supporting power-aware simulation have been extended to more advanced low-power design features, such as multiple-voltage designs and dynamic voltage scaling. Power-aware simulators need to understand the voltage levels to be able to accurately simulate the behavior. In addition, the expectation of power-aware simulation is not only to capture design bugs but also to get an early estimation of the power consumption.

Functional safety verification

Fault simulation is the main vehicle for evaluating the robustness of a product's safety mechanism against unexpected errors. Faults are injected into the design components to emulate the unexpected hardware errors. A safety module usually has a functional unit and a checker unit. When a fault is injected, the simulator outputs whether the fault can propagate to the functional output and the checker output. If a single fault can propagate to the functional output and cannot be detected by the checker, it means that the fault can propagate to the system without recovery, which is a single point of failure. Single point of failure is not tolerable for safety critical SoCs. Currently, most fault simulators support fault models including stuck-at faults, Single-Event-Upset and Single-Event-Transient faults. These fault models might be not sufficient to model realistic hardware errors without incorporating postlayout information. While post-layout information can enable more accurate fault modeling and fault distribution, it increases the complexity drastically and is not available until a late design stage. Even with a comprehensive fault model, it is challenging to inject faults at the right point in time and the right spot in the design to discover safety blind-spots. Currently, fault injection is usually performed randomly at the unit level and directed at the system level. Besides the fault injection in the digital part, fault injection in the analog part is also important and yet not well defined and supported. Echoing with the trend of hardware/software co-verification, yet another requirement for functional safety verification is to simulate the effects of erroneous execution of firmware code, which is not well supported by vendor tools.

Test generation and data analytics

Generation of high-quality tests is the driving force of the simulation-based verification to expose design errors. In microprocessor verification, constrained-random test generation has been widely adopted, where the verification engineers write test templates that are fed into the test generators to produce tests. The test generator infrastructure can be reused from one generation of processor design to another since they follow almost the same ISA. Test generation infrastructure in SoC verification used to be less reusable, where constrained-random test generation is often done during simulation and thus is bound to a specific testbench. We are delighted to see that there has been a paradigm shift in recent years to separate the test generation from the simulation testbench, which is the first step to increased reusability. Due to the aggressive scheduling, it is not possible to verify all aspects of an SoC. The test development of SoC verification is mainly driven by the intended use case scenarios of the SoC. Since the development of a test scenario requires knowledge and could be expensive, it is important that such a test scenario can be reused. An increasingly popular practice is to capture test scenarios in a test generation model and then to map the scenarios to tests filled with details at different levels of abstraction. There has also been an initiative (Portable Stimulus Working Group) to standardize the description language of the scenarios to maximize reusability [6].

Besides resuability, the most important and yet challenging task in test generation is to improve the efficiency of generating high-quality tests, where test quality is usually measured by coverage metrics. We call this task functional test content optimization. In constrained-random test generation, this means coming up with constraints to guide the directions of test generation. In a traditional verification flow, the acquisition of knowledge to enable such improvement is carried out mostly by manual learning, which requires a lot of nontrivial manual efforts. Figure 2 illustrates this process. The test generator *Gen* produces a set of tests *T*, while *Gen* can either be a verification engineer or a constrained-random test generator. The tests are applied to the simulator *Sim* to obtain the result *R*. The result *R* is evaluated through an evaluation step *Eval* and the important

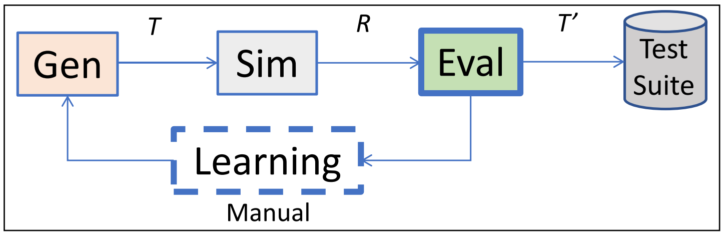


Figure 2. Existing manual learning in functional test content optimization [39].

tests T' are selected to be added into the test suite. An important test could be the one that excites a bug or provides coverage of some difficult-to-hit coverage events. When the result is not satisfactory, some aspects of the test generation process (e.g., the test template) are refined to produce new tests. The refinement is usually based on manual learning from the simulation result R and the information collected through step *Eval*.

In recent years, there have been a plethora of research on leveraging data mining (or in a more general term, data analytics) techniques to improve the quality and efficiency of test generation for coverage closure [16], [17], [24], [25], [28], [38]. Data mining is the process of extracting implicit, nontrivial, previously unknown, and potentially useful information from large data sets. In general, the information extracted by data mining can be classified into three main categories:

- descriptive: quantitatively describing the main characteristics of a data set;
- predictive: analyzing current and historical data to make predictions about future trends;
- prescriptive: making predictions and then suggesting decision options to take advantage of the predictions [11].

In functional test content optimization, the results from data analytics are usually used as prescriptive information. We introduce how data mining can be useful for functional test content optimization and the basic principles and considerations of applying data mining in functional verification.

Figure 3 shows two data mining components that can be added into the existing function test content optimization flow. First, a filter *Filter* can be added between the generator and the simulator. The filter plays the role of an oracle which can predict if a test can be effective for achieving the coverage goal. The

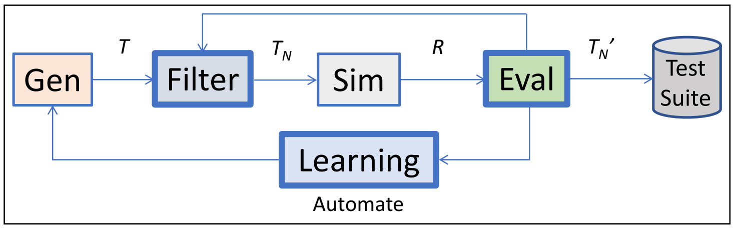


Figure 3. Two data mining components added in functional test content optimization [39].

objective is to filter out ineffective tests and thus only apply effective tests T_N for the simulation. An important assumption for this approach to be effective is that the cost of the simulation is much higher than the cost of the test generation. This approach does not directly improve the test generation but the efficiency of applying the tests in simulation.

The second component intends to automate the learning process for refining the aspects of test generation *Gen*. The learning results should be interpretable and actionable either by human or machine. The improvements from this approach can be two-fold:

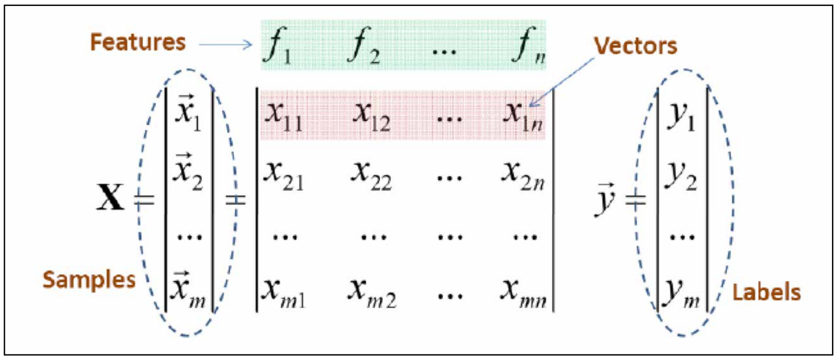
- for a coverage event that has been covered by only a few tests, the learning can be used to obtain more tests to increase the coverage frequency;
- for a coverage event that is not yet covered, the learning can be used to increase the chance of generating tests to cover the point.

To build practical data mining applications for functional test content optimization, one needs to formulate the target problem as one that can be solved by data mining and machine learning algorithms. Figure 4 illustrates a typical data set seen by a machine learning algorithm. When $\vec{y}$ is present, i.e., there is a label for every sample, it is called *supervised* learning. In *supervised* learning, if each y_i is a categorized value, it is a classification problem. If each y_i is a continuous value, it becomes a *regression* problem. When $\vec{y}$ is not present and only X is present, it is called *unsupervised* learning. When some (usually much fewer) samples are with labels and others have no label, the learning is then called *semisupervised* [15]. Interested readers can refer to [40] for a more detailed discussion.

To formulate a learning problem in functional verification, the first set of important questions concern the definition of a sample. For example, a sample could be an assembly test program. Alternatively, a sample could also be several consecutive instructions and an assembly program can be broken into several samples. Each sample is encoded with n features $f_{1,\ldots,}$ f_n. Hence, the characteristics of each sample are described as a feature vector $\vec{x}_i$. For a data mining algorithm to succeed, it is foremost critical to define the feature set with incorporated domain knowledge. This process is called feature engineering and coming up with good features is difficult, time-consuming, and requires expert knowledge and many iterations.

After defining the feature set, the next set of questions concern the learning approaches. It depends on the data availability and the application scenarios. Suppose we would like to learn why certain tests can hit certain coverage events. If we have two categories of tests, one that covers the events and one that does not, then binary classification can usually be applied. However, if there are no tests covering the events, outlier analysis would be a better option, as the learned model can be used as the *Filter* component in Figure 3. Even with a learning approach decided, there can be many algorithms to choose from. Different algorithms can produce learning models in different forms. The representation of a learning model can be based on rules, trees, equations, or a collection of samples. It depends on how the model is applied. For example, if the model is first to be interpreted by a person, then a complex model would not be helpful while a rule or a tree model might be more suitable. Many algorithms might have variations and parameters to decide on. In addition, often preprocessing of the data set such as dimension reduction and feature selection is needed to facilitate the learning [39]. An empirical observation is that the definition of the feature set plays a more important role than the algorithms for a data mining application.

After applying the learning algorithms, it is critical to validate the learned results (models) to make sure that they do not *overfit* the training data. This is often done by cross validation, where a relatively small portion of data is preserved as the validation data set to ensure that the model performs similarly on both training data and validation data. In functional test content optimization, further validation can be carried out via application of the learning result. For example, if a rule is learned for hitting a particular event, evaluation of the model can mean modifying the test templates according to the rule and observing the coverage of the events in simulation of the resulting tests. If we observe increased coverage, it means that the learning result is meaningful.



$$\mathbf{X} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \cdots \\ \vec{x}_m \end{bmatrix} = \begin{array}{|cccc|} f_1 & f_2 & \cdots & f_n \\ x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{array} \qquad \vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_m \end{bmatrix}$$

Figure 4. Typical dataset in a machine learning problem [40].

Targeted formal methods

Formal verification is the process of proving or disproving the correctness of a system with respect to a certain formal specification or property, using formal methods of mathematics. Often, verification is concerned with two common forms of properties:

- *safety* property, which ensures something bad does not happen;
- *liveness* property, which ensures something good eventually does happen.

The formal specification or property is described using a formal language such as Property Specification Language (PSL) and SystemVerilog Assertion (SVA) and then a formal method/tool can determine either the specification/property holds true or find a counterexample showing the circumstances where the specification/property is violated. Notwithstanding formal methods can offer thorough verification with respect to the pertinent specification, there have been two major obstacles to replace simulation with formal methods for verifying complex SoCs:

(1) *Tool scalability:* formal methods suffer from the state explosion problem, namely, the size of the problem that formal tools are dealing with grows exponentially as the number of state variables in the system increases [18]. The overwhelming growth of complexity limits the applicability of formal methods to large industrial designs.

(2) *User friendliness:* the application of formal methods often requires a deep understanding in underlying principles of the tool and familiarity with the design-to-verify. This is partly caused by the tool scalability issue since verification engineers often need to manually apply techniques such as blackboxing, abstraction and refinement, and assume-guarantee reasoning to reduce the complexity. In-depth understanding of the reset and clocking schemes is often a prerequisite for setting up the formal

verification environment. In addition, commercial formal tools are currently limited to supporting digital circuits and cannot be readily applied to AMS circuits.

Commercial formal tools leverage a combination of different formal engines to solve a problem. In recent years, advances in core technologies of formal engines have been focused on efficient SAT and SMT solvers to improve tool scalability. On the other hand, what has made formal methods more applicable in reality is the rapid growth of end-to-end applications of formal methods to solve target problems with well-defined input, often called "Automatic Formal Apps." This shows a paradigm shift from the traditional way of performing formal verification where engineers setup environments and describe formal properties in a low-level language. Targeted formal methods essentially accept the specifications in a higher-level representation such as spreadsheets or XML, automatically generate properties, invoke formal engines, and finally generate human interpretable reports. This significantly reduces the manual efforts and lowers the barriers for applying formal methods to solve verification problems. We review some of the targeted areas where Automated Formal Apps have prospered.

Secure information flow analysis

Among the most common SoC security requirements are *confidentiality* and *integrity* of on-chip assets. Proper access to the assets must be assured by the enforcement of information flow policies, which is one of the most important aspects of modern computer security, and yet is also one of the hardest to get correct in implementation [37]. Confidentiality requires that an asset cannot be copied or stolen, which is essential for assets such as passwords and cryptographic keys. Integrity requires that an asset is defended against modification, which is essential for some of the on-chip root secrets on which the rest of the system security is based, e.g., configuration firmware, and for the security software once it is running.

Confidentiality and integrity can be expressed as secure information flow properties, the research of which originated from software verification [33] and made its way to hardware verification [34]. Secure information flow concerns whether the information in a secure world has an impact on that in an insecure one, and vice versa. Confidentiality of an asset means that there is no information flow from where the asset is contained to untrusted locations. Integrity of an asset indicates that there is no information flow from untrusted locations to where the asset resides.

Unlike safety and liveness properties, secure information flow property belongs to the category of hyperproperties [19], which cannot be described by PSL or SVA. Therefore, for secure information flow properties to be proved/disproved by commercial formal tools, they are usually translated to safety properties in a transformed system. An example transformation is self-composition, where the transformed system comprises two copies of the original one [36]. A violation of secure information flow properties thus can be represented as a pair of traces in respective copies of the original system. The difference between the traces essentially shows the sensitization path of information leakage or contamination.

Connectivity verification

SoCs are designed by connecting and integrating hundreds of IP blocks. An average SOC may have connections in the order of tens of thousands. While the process of connecting and integrating IPs is usually accomplished by automated design tools in a "correct by construction" manner, errors can still creep in due to unclear or erroneous specifications, the addition of low-power structures, built-in self-test and JTAG support, or downstream changes/engineering change orders that were not fully propagated [22]. There are two types of connectivity in SoC designs:

- static connection, consisting of simple hookup of the inputs and outputs of different IPs;
- dynamic, or functional integration, where, besides the pure static connection, a temporal and a functional dimension needs to be taken into account due to the practice of hardware resource sharing [32].

A typical example of dynamic connectivity is the pin multiplexing scheme, which allows the internal IP blocks to dynamically share the routes to I/O pads of a limited number of external pins. Because of the tremendous complexity caused by the large number of connections and complex multiplexing policies, there is a need for thorough verification of

SoC connectivity and formal methods are deemed as a nature fit.

In formal verification of SoC connectivity, the specifications are usually captured in formats such as spreadsheets and XML files. Then the tool will read the specification and automatically generate the properties to run formal proof. One challenge of this practice is to come up with a standardized specification scheme that is expressive enough to capture various connection schemes. Another challenge is to manage the proof process of tens of thousands of properties, which often requires parallel execution of formal engines on the compute farm.

Besides formally verifying the connectivity of the design with respect to the specification, another application is to extract the high-level connectivity specification from a design where the specification is unavailable [4]. The extracted specification can be used for design reviews. Moreover, it is not uncommon that an SoC design will be changed for feature enhancement or performance improvement without intention to change the connectivity. Under this circumstance, the extracted specification can be used to formally verify that the connectivity is not messed up by the design change.

Coverage unreachability analysis

SoC designs nowadays heavily reuse IP blocks in order to maximize productivity. This trend results in the fact that certain features of a reused IP are disabled in a given SoC as they are not needed. Under this circumstance, the portion of RTL code implementing the features becomes dead code. The coverage of the dead code needs to be waived during coverage analysis as those code blocks will never be exercised. In the process of coverage closure, verification teams need to decide whether a coverage hole should be waived or should be given more resources to close it. Given the large number of IP blocks and often lack of accurate documentation, waiver of dead code is a tedious and labor-intensive task. However, it is a niche problem for applying formal methods. After identification of RTL code blocks that were not covered by extensive simulation, formal properties concerning exercising those code blocks can be generated. If the formal tool decides that the code blocks cannot be exercised, they are labeled as dead code. Like formal verification of connectivity, the challenge here is to manage the proof process of a large number of properties.

One might ask if the same analysis can be used to generate tests that cover the coverage holes. However, it is much more trickier than detecting dead code. When the formal tool finds a trace where the target code block is exercised, the trace might not be a valid one as there could be missing constraints in the proof process. To be able to use the same analysis to generate tests, environmental constraints need to be accumulated through the simulation-based verification process. This requires a lot of manual efforts and there are research opportunities to apply data mining methods to learn those constraints.

Register verification

IP blocks in an SoC contain many memory-mapped registers that can be accessed by system or user programs to configure the IP and check status. The implementation complexity of memory mapped registers is relatively low compared to other design elements. The challenge is that there are an excessive number of memory mapped registers with various access policies such that manually verifying them by directed or lightly randomized tests is a tedious and error-prone job [26]. Complicating the problem is the existence of advanced features such as register aliasing and register remapping under different operational modes. Formal verification of register access is motivated by similar reasons with formal verification of connectivity and shares similar characteristics and challenges.

Another practical challenge in register verification is the lack of a reliable specification. It is not uncommon that the specification documents omit details such as when a memory-mapped register can be modified by internal operations [26]. Verification engineers often rely on trial-and-error to determine the undocumented behavior. Another application of formal methods in register verification is to automate this manual exploration process by formulating formal properties regarding access policies and checking the correctness. The challenge here is to have comprehensive presumptions of access policies that cover those advanced features.

We have discussed the complexity and challenges in modern hardware verification, and described some recent trends in verification technology to address these challenges. Many of the challenges are reminiscent of the past, e.g., lack of specification, tool scalability, and challenges in security validation. Nevertheless, as the ecosystem of computing devices changes, the problems appear in different forms and priorities.

As studies indicate, verification is getting to become a bottleneck both in effort and in cost to the entire design process. Addressing tool scalability, targeting verification to specific needs, etc., are steps to alleviate the process. However, such efforts cannot be expected to go far enough. In order to significantly reduce the cost, it is important for systems to be designed with verification in mind rather than verification imposed post facto on an already designed system. In fact, that is already happening for some aspects of verification, particularly security and post-silicon. However, to address the challenge of scalability and robustness in verification, it is incumbent that we take stronger and more comprehensive steps in integrating functional verification needs tightly into the design process. ■

References

[1] *1801–2015—IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems*. Accessed May 30, 2017. [Online]. Available: https://standards.ieee.org/findstds/standard/1801-2015.html.

[2] *Certus^{TM} Silicon Debug*. Accessed May 30, 2017. [Online]. Available: https://www.mentor.com/products/fv/certus- silicon-debug.

[3] *HAPS ProtoCompiler.* Accessed May 30, 2017. [Online]. Available: https://www.synopsys.com/verification/prototyping/haps/haps-protocompiler.html.

[4] *JasperGold Connectivity Verification App.* Accessed May 30, 2017. [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/connectivity-verification-app.html.

[5] *Palladium Z1 Enterprise Emulation System*. Accessed May 30, 2017. [Online]. Available: www.cadence.com.

[6] *Portable Stimulus Specification Working Group.* Accessed May 30, 2017. [Online]. Available: http://www.vhdl.org/activities/working-groups/portable-stimulus.

[7] *Si2 Common Power Format.* Accessed May 30, 2017. [Online]. Available: http://projects.si2.org/openeda.si2.org/project/showfiles.php?group_id=51.

[8] *Veloce2 Emulator*. Accessed May 30, 2017. [Online]. Available: https://www.mentor./-com-/products/fv/emulation-systems/veloce.

[9] *Verilog-AMS Language Reference Manual*. Accessed May 30, 2017. [Online]. Available: http://accellera.org/images/downloads/standards/v-ams/VAMS-LRM-2-4.pdf.

[10] *Zebu*. Accessed May 30, 2017. [Online]. Available: http://www.synop/-sys.com/tools/verification/hardware-verification/emulation/Pages/default.aspx.

[11] M. Arar et al., "The verification cockpit—Creating the dream playground for data analytics over the verification process," in *Proc. 11th Int. Haifa Verification Conf., HVC 2015,* Haifa, Israel, pp. 51–66.

[12] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. Amsterdam, The Netherlands: IOS, vol. 185, pp. 825–885, 2009.

[13] R. Bell, "Introduction to IEC 61508," in *Proc. 10th Australian Workshop Safety Critical Syst. Software – SCS '05*. Darlinghurst, Australia, vol. 55, 2006, pp. 3–12.

[14] J. Bhadra, M. S. Abadir, L. Wang, and S. Ray, "A survey of hybrid technqiues for functional verification," *IEEE Des. Test Comput.*, vol. 24, no. 2, pp. 112–122, 2007.

[15] O. Chapelle, B. Schlkopf, and A. Zien, *Semi-Supervised Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 2010.

[16] W. Chen, N. Sumikawa, L.-C. Wang, J. Bhadra, X. Feng, and M. S. Abadir, "Novel test detection to improve simulation efficiency: A commercial experiment," in *Proc. Int. Conf. Comput. Aided Des.*, New York, NY, USA, 2012, pp. 101–108.

[17] W. Chen, L.-C. Wang, J. Bhadra, and M. Abadir, "Simulation knowledge extraction and reuse in constrained random processor verification," in *Proc. ACM/IEEE Des. Autom. Conf.*, 2013.

[18] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the state explosion problem in model checking," *in Informatics—10 Years Back. 10 Years Ahead.* R. Wilhelm, Ed. London, UK: Springer-Verlag, 2001, pp. 176–194.

[19] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, pp. 1157–1210, Sept. 2010.

[20] H. Foster, "Trends in functional verification: A 2014 industry study," in *Proc. Des. Autom. Conf. 2015*, 2016, pp. 48:1–48:6.

[21] A. Gupta, "Formal hardware verification methods: A survey," *Form. Methods Syst. Des.*, vol. 2, no. 3, pp. 151–238, Oct. 1992.

[22] Z. Hanna, “Challenging problems in industrial formal verification,” in *Proc. 14th Conf. Form. Methods Comput. Aided Des.,* Austin, TX, 2014, p. 1:1.
[23] *Road Vehicles–Functional Safety*, ISO, 2011.
[24] V. Kamath, W. Chen, N. Sumikawa, and L.-C. Wang, “Functional test content optimization for peak-power validation-an experimental study,” in *Proc. IEEE Int. Test Conf.*, Anaheim, CA, USA, Nov. 2012, pp. 1–10.
[25] Y. Katz, M. Rimon, A. Ziv, and G. Shake, “Learning microarchitectural behaviors to improve stimuli generation quality,” in *Proc. ACM/IEEE Des. Autom. Conf.*, 2011, pp. 848–853.
[26] N. Kim, J. Park, B. Min, and W. Park, “Register verification: Do we have reliable specification?” in *Proc. Des. Verification Conf.*, 2013.
[27] X. Li, C. Kashyap, and C. J. Myers, “Guest editors’ introduction challenges and opportunities in analog/mixed-signal CAD,” *IEEE Des. Test*, vol. 33, no. 5, pp. 5–6, Oct. 2016.
[28] L. Liu, D. Sheridan, W. Tuohy, and S. Vasudevan, “Towards coverage closure: Using GoldMine assertions for generating design validation stimulus,” in *Proc. Des., Autom. Test Europe Conf. Exhi.*, 2011, pp. 1–6.
[29] P. Mishra, R. Morad, A. Ziv, and S. Ray, “Post-silicon validation in the SoC era: A tutorial introduction,” *IEEE Des. Test Comput.*, vol. 34, no. 3, pp. 68–92, Jun. 2017.
[30] S. Ray, I. Harris, G. Fey, and M. Soeken, “Multilevel design understanding: From specification to logic,” in *Proc. ICCAD*, 2016, p. 133.
[31] S. Ray, Y. Jin, and A. Raychowdhury, “The changing computing paradigm with internet of things: A tutorial introduction,” *IEEE Des. Test*, vol. 33, no. 2, pp. 76–96, 2016.
[32] S. K. Roy, “Top level SOC interconnectivity verification using formal techniques,” in *Proc. 2007 8th Int. Workshop Microprocessor Test Verification*, Dec. 2007, pp. 63–70.
[33] G. Smith, “Principles of secure information flow analysis,” *Malware Detection*, M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, Eds. Boston, MA, USA: Springer, 2007, pp. 291–307.
[34] P. Subramanyan and D. Arora, “Formal verification of taint-propagation security properties in a commercial SoC design,” in *Proc. Des., Autom. Test Europe Conf. Exhi.*, 2014, pp. 1–2.
[35] M. Talupur, S. Ray, and J. Erickson, “Transaction flows and executable models: Formalization and analysis of message passing protocols,” in *Proc. Form. Methods Comput. Aided Des.*, 2015, pp. 168–175.
[36] T. Terauchi and A. Aiken, “Secure information flow as a safety problem,” in *Proc. 12th Int. Conf. Static Analysis.* Berlin, Germany: Springer-Verlag 2005, pp. 352–367.
[37] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, “Complete information flow tracking from the gates up,” in *Proc. 14th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, New York, NY, USA, 2009, pp. 109–120.
[38] I. Wagner, V. Bertacco, and T. Austin, “Microprocessor verification via feedback-adjusted Markov models,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 26, no. 6, pp. 1126–1138, Jun. 2007.
[39] L. C. Wang, “Data mining in functional test content optimization,” in *Proc. 20th Asia South Pacific Des. Autom. Conf.*, 2015, pp. 308–315.
[40] L.-C. Wang and M. S. Abadir, “Data mining in EDA—Basic principles, promises, and constraints,” in *Proc. 51st Ann. Des. Autom. Conf. (ACM)*, New York, NY, USA, 2014, pp. 159:1–159:6.
[41] B. Wile, J. Goss, and W. Roesner, *Comprehensive Functional Verification: The Complete Industry Cycle.* San Francisco, CA, USA: Morgan Kaufmann, 2005.

Wen Chen is a Principal Engineer at NXP Semiconductors. His research interests include functional verification of microprocessor and SoC, data mining, and security. He received an MS in computer science and engineering from the University of Michigan, Ann Arbor, and a PhD in electrical and computer engineering from the University of California, Santa Barbara. He is a Member of the IEEE.

Sandip Ray is a Senior Principal Engineer at NXP Semiconductors, where he leads R&D on security validation for automotive and Internet-of-Things applications. His research interests include developing correct, dependable, secure, and trustworthy computing through cooperation of specification, synthesis, architecture, and validation technologies. He received a PhD from the University of Texas at Austin. He is a Senior Member of the IEEE.

Jayanta Bhadra is the Director of World-wide Systems and Verification Design Enablement team at NXP Semiconductors. His research interests include hardware verification, mathematical logic, and security related research. He received a PhD in electrical and computer engineering from the University of Texas at Austin. He is a Senior Member of the IEEE.

Magdy Abadir is currently on the board of directors of Helic and also serves as Vice President of Corporate Marketing. His research interests include microprocessor test and verification, test economics, and DFT. He received a BS with honors in computer science from Alexandria University, Egypt, an MS in computer science from the University of Saskatchewan, and a PhD in electrical engineering from the University of Southern California. He is an IEEE Fellow for contributions he made to microprocessor test and verification.

Li-C Wang is a Professor in the Department of Electrical and Computer Engineering, University of California, Santa Barbara. His research interests include microprocessor test and verification, statistical methods for timing analysis, speed test and performance validation, and applications of data mining and statistical learning in EDA. He received an MS in computer science and a PhD in electrical and computer engineering from the University of Texas at Austin.

■ Direct questions and comments about this article can be sent to Wen Chen, NXP Semiconductors, Austin, TX, USA; e-mail: wen.chen@nxp.com.