A Survey on Assertion-based Hardware Verification

HASINI WITHARANA, University of Florida, USA
YANGDI LYU, Hong Kong University of Science and Technology, China
SUBODHA CHARLES, University of Moratuwa, Sri Lanka
PRABHAT MISHRA, University of Florida, USA

Hardware verification of modern electronic systems has been identified as a major bottleneck due to the increasing complexity and time-to-market constraints. One of the major objectives in hardware verification is to drastically reduce the validation and debug time without sacrificing the design quality. Assertion-based verification is a promising avenue for efficient hardware validation and debug. In this paper, we provide a comprehensive survey of recent progress in assertion-based hardware verification. Specifically, we outline how to define assertions using temporal logic to specify expected behaviors in different abstraction levels. Next, we describe state-of-the art approaches for automated generation of assertions. We also discuss test generation techniques for activating assertions to ensure that the generated assertions are valid. Finally, we present both pre-silicon and post-silicon assertion-based validation approaches that utilize simulation, formal methods as well as hybrid techniques. We conclude with a discussion on utilizing assertions for verifying both functional and non-functional requirements.

CCS Concepts: • **Hardware** → **Functional verification**; **Assertion checking**; **Post-manufacture validation and debug**; **Bug detection, localization and diagnosis**;

Additional Key Words and Phrases: Hardware Verification, Post-silicon debug, Assertion-based validation, Assertion generation, Test generation

ACM Reference Format:
Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. 2022. A Survey on Assertion-based Hardware Verification. *ACM Comput. Surv.* 1, 1 (January 2022), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

We are dependent on computing systems to provide us a comfortable lifestyle. Such systems consist of hardware, software as well as application-specific input/output capabilities. Hardware is the brain behind these computing systems to provide performance, reliability, energy efficiency as well as security requirements. Even simple Internet-of-Things (IoT) devices include one or more System-on-Chip (SoC) hardware that consists of millions of transistors. Due to the increasing complexity of modern electronic systems, functional validation is widely acknowledged as a major challenge in SoC design methodology.

As shown in Figure 1, according to the 2020 Wilson research group functional verification study [52], 51% of development effort (cost) in both ASIC and FPGA-based systems were spent on

Authors' addresses: Hasini Witharana, University of Florida, Gainesville, FL, 32611, USA, witharana.hasini@ufl.edu; Yangdi Lyu, Hong Kong University of Science and Technology, Guangzhou, Guangdong, 511458, China; Subodha Charles, University of Moratuwa, Moratuwa, 10400, Sri Lanka; Prabhat Mishra, University of Florida, Gainesville, FL, 32611, USA.

0360-0300/2022/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

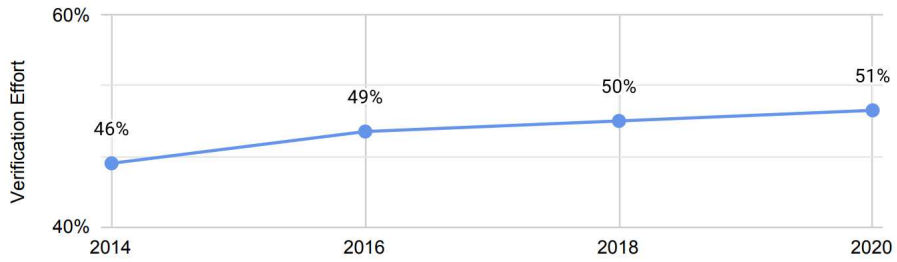


Fig. 1. Average project time spent in functional verification. In 2020, the average verification effort is about 51% of the overall development effort (cost and time) [52].

verification. In spite of such a huge investment in verification, the study also reveals that a vast majority of systems fail the first time (only 32% of electronic systems have their first silicon success). To address this fundamental bottleneck in terms of both verification/debug time and final product quality, the semiconductor industry employs advanced and effective verification techniques into their existing SoC design methodology.

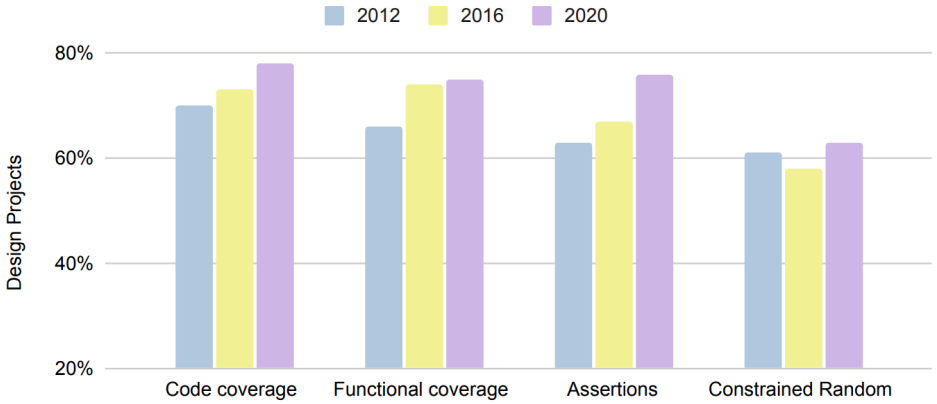


Fig. 2. ASIC functional verification trend. In 2020, more than 75% ASIC designs utilized assertion-based validation [52].

The study also highlights that both ASIC (Figure 2) and FPGA (Figure 3) verification use an effective combination of code coverage, functional coverage, assertion-based verification (ABV), and validation using constrained-random test patterns. These figures provide two important insights: (i) a vast majority of ASIC designs (more than 75%) and almost 50% of FPGA design projects used assertion-based validation in 2020, and (ii) there is a steady increase in the adoption of assertion-based validation over the years. These observations suggest that ABV is a critical component today for functional validation, and is expected to remain critical in the foreseeable future. The remainder of this section is organized as follows. First, we provide an overview of traditional hardware design methodology. Next, we outline the sources of potential bugs and vulnerabilities during hardware design flow. Finally, we outline assertion-based validation for detecting these bugs and vulnerabilities.

1.1 Overview of Hardware Design Flow

Figure 4 shows a typical hardware design methodology. There are multiple abstraction levels including pre-silicon (before fabrication) and post-silicon (after fabrication). Depending on the

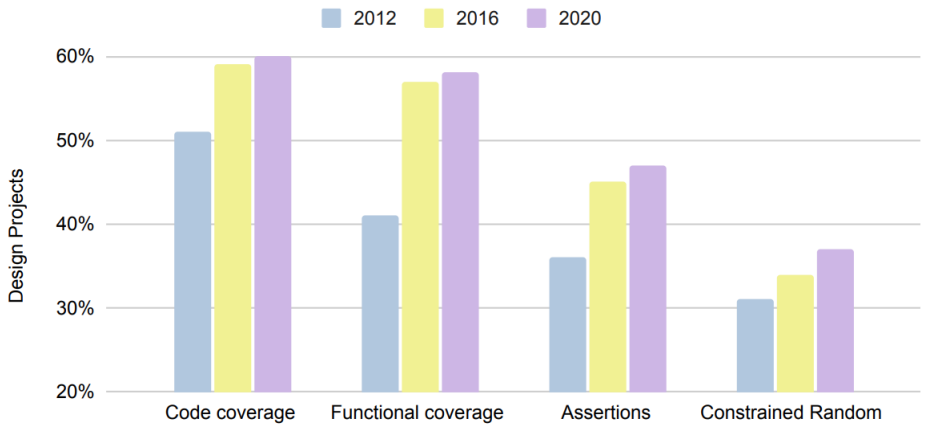


Fig. 3. FPGA functional verification trend. Almost 50% FPGA designs utilized assertion-based validation in 2020 [52].

complexity and types of hardware designs, it starts at different abstraction levels. In this figure, the high-level architecture of the design is specified in Transaction Level Modeling (TLM). SystemC TLM is a widely used form of architecture specification for processors as well system-on-chip (SoC) designs. TLM abstraction is ideal for architectural exploration as well as specification validation [32, 34, 35]. The TLM specification can be used to generate RTL (Register Transfer Level) models, which is closer to actual silicon by adding interfacing and timing on top of the TLM models. VHDL and Verilog are typically used to write RTL models of the design. Pre-silicon validation can be done for RTL designs to verify the behaviour or to ensure that the RTL implementation satisfies the TLM specification. Assertions can be added in both TLM and RTL designs to perform assertion-based validation. The RTL implementation can be synthesized to gate-level netlist. Synthesized assertions (if any) will work as post-silicon checkers. A typical VLSI design flow consists of many more steps including layout and fabrication. The fabricated design (silicon) can be used for post-silicon validation. The checkers in silicon can also be used for in-field debugging of the final product.

Validation reuse is a critical consideration in SoC design methodology to save overall validation effort (cost and time). For example, test patterns generated for activating gate-level assertions can be used for activating post-silicon checkers. Similarly, pre-silicon assertions can be synthesized to post-silicon checkers. Clearly, synthesizing all the assertions will provide 100% coverage at post-silicon. Unfortunately, it can lead to violation of design constraints such as area, power and performance budget. There are promising approaches to synthesize profitable pre-silicon assertions to trade-off post-silicon coverage versus harwdare overhead [48].

1.2 Sources of Hardware Bugs and Vulnerabilities

The goal of SoC hardware validation is to ensure that the implementation satisfies both functional behaviors and non-functional (e.g., energy, security, etc.) requirements. There are a wide variety of validation objectives including detection of functional bugs, timing errors, security vulnerabilities, etc. These errors and vulnerabilities can come from a wide variety of sources, including:

- The design goes through various phases including specification, synthesis, integration, fabrication, etc. Many parties and tools are involved in the long process of hardware design. This can lead to errors and vulnerabilities due to design mistakes and/or buggy CAD tools.
- With the ever-increasing complexity and decreasing size of hardware designs, the outsourcing and integration of hardware Intellectual Property (IP) has become a new trend. However,

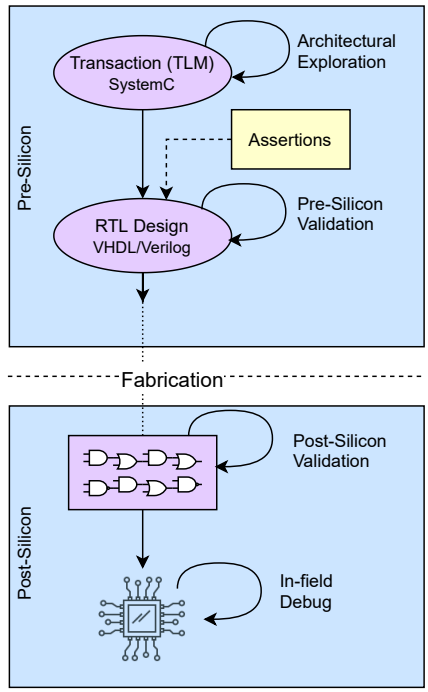


Fig. 4. Hardware Design Flow

such IPs from potentially untrusted third-party vendors and untrusted manufacturers can introduce errors and vulnerabilities.

- The running environments of these devices are heterogeneous and possibly connected and unprotected. Vulnerabilities in one device may open a backdoor to other devices. In addition, undetected errors, unverified hidden states/transitions, and soft errors in hardware can lead to more vulnerabilities during run-time.

These errors and vulnerabilities in hardware cannot be detected by tools in the software domain, such as anti-virus tools. They are also hard to fix after deployment. Firmware patching or built-in re-configurable primitives can mask some of the vulnerabilities, but may not fix all of them. Therefore, a highly automated and scalable hardware validation scheme is a must to detect and remove as many bugs and vulnerabilities as possible.

1.3 Overview of Assertion-based Verification

One problem in hardware validation is how to increase the controllability and observability to reveal internal errors and bugs. Controllability represents the ability to control internal signal, and observability represents the ability to see the state of the design. The embedded assertions can catch any unexpected behavior which increases the observability of internal activities inside the design. For example, an assertion can check that the output of an adder is equal to the sum of its inputs even though the implementation is inside the execution stage of a CPU. Any bug in the design that violates the predefined properties in assertions can be easily detected. The observability of internal states enables faster localization of errors, which reduces the overall validation time significantly. There are several approaches to generate tests to activate assertions that can reveal the internal states. As shown in Figure 5, ABV can improve both controllability and observability, and enable faster debug. When an assertion fails during simulation, it will provide enough information to the

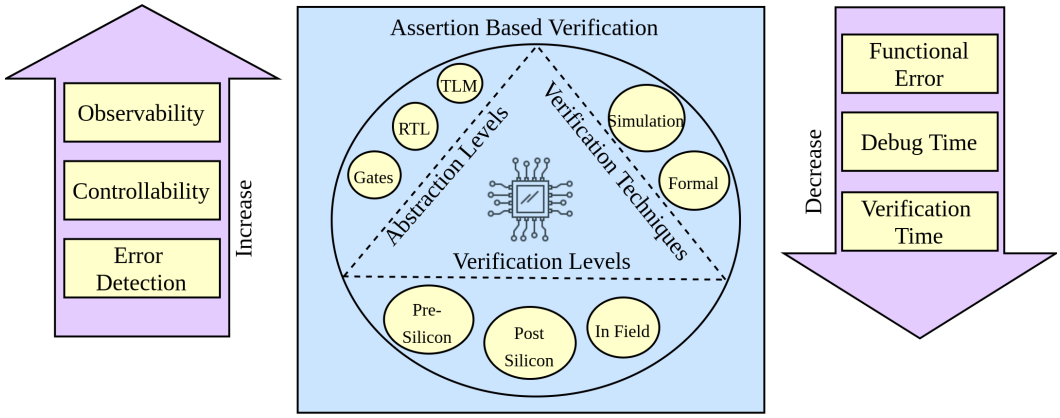


Fig. 5. Overview of Assertion Based Verification (ABV) in hardware designs. It can be applied across abstraction levels and validated using both simulation and formal methods. ABV improves observability and controllability. As a result, it provides drastic reduction in overall verification and debug effort.

designer to start fixing the problem. In contrast, in the absence of assertions, it may take hours or days to even find out the reason for a failure. As a result, effective use of assertions can drastically reduce the verification and debug time.

Example 1: Listing 1 shows two sample assertions in the arbiter design. Assertions can be classified as immediate and concurrent based on their embedding in the design. For example, *assert (req2==ack2)* is an immediate assertion in Listing 1, and it gets activated immediately after the execution of the statement "*ack2=req2*" if $ack2$ is not equal to $req2$. Similarly, *assert property(ack1 |-> ¬ ack2)* is a concurrent assertion in Listing 1 since it runs concurrently with other modules, and it gets activated when both $ack1$ and $ack2$ become true at the same time. □

Assertions can be utilized for pre-silicon validation as well as post-silicon debug. Assertions can be embedded at different abstraction levels including Transaction-Level Model (TLM) and Register-Transfer Level (RTL). Compared to RTL models, TLM is more abstract and faster in simulation. Therefore, TLM is more suitable for the validation of large designs and hardware/software co-design. There are many challenges in applying assertion-based validation. Here we outline four major considerations.

(1) How to generate enough assertions that are able to both validate the correct functionality of the design and capture all the potential vulnerabilities. It is the main challenge in assertion-based validation due to the complex validation space. For example, to validate the correct behavior of the design, all the states and transitions of its finite state machine should be considered. However, the number of such transitions could be enormously large considering the number of unrolled cycles.
(2) How to generate effective test patterns to reveal vulnerabilities in the design, i.e., activate the assertions if such vulnerabilities exist. Traditional test generation approaches using random/constrained-random tests are not effective to activate a given assertion. Automated test generation approaches, such as using formal methods, can lead to state explosion for large designs.
(3) How to reuse assertions across different abstraction levels. Since the design at different abstraction levels is maintained by a different group of engineers, management of assertions

as well as design modifications are typically manual in nature. Reuse of assertions across different abstraction levels is promising to minimize the wasted efforts.

(4) What subset of assertions should be embedded in the silicon. The assertions/checkers take additional area and power overhead in the silicon. To meet the constraints of different parameters and also to ensure the correct behavior of the design, it is a major challenge to select a small subset of assertions that will be embedded in the silicon without negatively impacting post-silicon debug.

Listing 1. Examples of concurrent and immediate assertions

```
module arb(clk, rst, req1, req2, ack1, ack2);
input clk, rst, req1, req2;
output ack1, ack2;
reg state, ack1, ack2;
always @ (posedge clk or posedge rst)
    if (rst)
        state <= 0;
    else
        state <= ack1;
always @ (*)
    if (state) begin
        ack1 = req1 & ~req2;
        ack2 = req2;
        assert(req2 == ack2) \\ Immediate Assertion
    end
    else begin
        ack1 = req1;
        ack2 = req2 & ~req1;
    end
assert property(ack1|->~ack2) \\ Concurrent Assertion
endmodule
```

The outline of the paper is as follows. Section 2 provides a background on temporal logic to illustrate the expressive power of assertions. Section 3 describes a wide variety of applications of assertions including functional validation as well as validation of non-functional requirements. Section 4 briefly introduces the methodology of this survey paper. Section 5 covers some state-of-the-art topics and methods in automated assertion generation. Section 6 presents different test generation techniques for activating assertions. Section 7 surveys assertion-based pre-silicon verification techniques. Section 8 describes assertion-based post-silicon debug methods. Finally, Section 9 shows a few directions for future research in assertion-based verification and concludes this paper.

2 TEMPORAL LOGIC

In this section, we briefly describe temporal logic [88] to highlight the expressive power of assertions. Temporal logic is widely used in property checking based formal verification as well as assertion-based validation. Temporal logic is succinct but expressive enough to represent most properties that a finite-state system needs to satisfy. Therefore, temporal logic is also the standard formal language to define assertions. Note that properties and assertions possess the same representation capability since they capture the behavior using temporal logic. To express different types of properties, a variety of temporal logic have been proposed. For example, Linear-time Temporal Logic (LTL) is able to describe a property along with a single execution. However, it lacks the ability to express other possible executions, and Computation Tree Logic (CTL) is proposed to solve the problem.

Property Specification Language (PSL) [1] and System Verilog Assertions (SVA) [2] are very popular assertion specification languages. Both PSL and SVA support LTL and CTL extended temporal assertions. In this section, we introduce three types of logic in LTL and CTL.

2.1 Propositional logic

Propositional logic allows us to reason about the truth or falsehood of logical expressions. A proposition is evaluated to be either true or false. Multiple propositions can be connected using logical operators to form a propositional formula. Table 1 shows a few commonly used logical operators. The table is sorted by the order of precedence, with '¬' having the highest precedence. Propositional formulas by themselves do not have a notion of timing. If assertions are written using propositional formulas only, they can be thought of as combinational assertions.

Table 1. Basic Logical Operators

Connective	Operation	Example
¬	not	$\neg P$ is false if P is true.
∧	and	$P \wedge Q$ is true if both P and Q are true. False otherwise.
∨	or	$P \vee Q$ is true if either P or Q is true. False otherwise.
→	implication	$P \rightarrow Q$ is false when P is true and Q is false. True otherwise
⇔	equivalent	$P \Leftrightarrow Q$ is true if P and Q have same truth values. False otherwise.

Example 2: Consider the sample RTL module in Listing 1 for property generation using temporal logic. The logic of the design is represented in the truth table as shown in Table 2. The first three columns of the table represent the inputs: $state$, $req1$ and $req2$, and the next 2 columns represent the outputs: $ack1$ and $ack2$. According to the truth table, $ack1$ and $ack2$ should not be activated together. In other words, $ack1$ and $ack2$ cannot have value 1 at the same time. This property can be represented logically using the implication operation such as $(ack1 \rightarrow \neg ack2)$. Another equivalent form of representing the property is $(\neg ack1 \vee \neg ack2)$. □

Table 2. Truth table for RTL module in Listing 1

Input			Output	
state	req1	req2	ack1	ack2
1	1	1	0	1
1	1	0	1	0
1	0	1	0	1
1	0	0	0	0
0	1	1	1	0
0	1	0	1	0
0	0	1	0	1
0	0	0	0	0

2.1.1 Conjunctive Normal Form (CNF). Every propositional formula can be converted to Conjunctive Normal Forms (CNF). Many proofs assume that the propositional formula is given in CNF. Given below is the generic form of CNF [38]:

$$(A_{11} \vee A_{12} \vee ... \vee A_{1n_1}) \wedge (A_{21} \vee ... \vee A_{2n_2}) \wedge ... \wedge (A_{m1} \vee ... \vee A_{mn_m}) \qquad \text{or} \qquad \bigwedge_i \bigvee_j A_{ij}$$

Here all A_{ij} terms are called *literal*. These are proposition or a negation of proposition. In CNF, conjunction ($\vee$) of the literals form a *clause*. For example, here $(A_{11} \vee A_{12} \vee ... \vee A_{1n_1})$ is a clause. Disjunction ($\wedge$) of the clauses forms the complete propositional formula in CNF.

2.1.2 Disjunctive Normal Form (DNF). Similar to CNF, every propositional formula can also be written as a Disjunctive Normal Form (DNF). The generic form of DNF is as follows:

$$(A_{11} \wedge A_{12} \wedge ... \wedge A_{1n_1}) \vee (A_{21} \wedge ... \wedge A_{2n_2}) \vee ... \vee (A_{m1} \wedge ... \wedge A_{mn_m}) \qquad \text{or} \qquad \bigvee_i \bigwedge_j A_{ij}$$

Similar to CNF, all A_{ij} terms are literals. In DNF, the disjunction of the literals forms a clause. Conjunction of the clauses forms the complete propositional formula in CNF.

2.2 Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) extends the propositional logic by introducing the notion of timing [16]. It can be used to describe a sequence of transitions along a path. Some of the temporal operators that are used to describe these transitions are given in Table 3, where p and q are propositions which can contain both logic operators and temporal operators. Each proposition is either true or false at a given time. For example, $F\,p$ represents that eventually, p will be true at some future state, as shown in Figure 6(a), where each node represents a state. Another example would be $p\ U\ q$ which represents that p is true at every preceding state before q is true, which is shown in Figure 6(b). Assertions with LTL is more powerful than combinational assertions. For example, the property ($work\ U\ done$) can ask a system to continuously work until the job is done, and the "always" operator G can be combined with "not" operator $\neg$ to ensure that some vulnerabilities never occur in a system.

Table 3. Basic temporal operators in LTL [39, 85]

Op.	Semantics	Description
$X\,p$	next	p is true in the next state of the path.
$G\,p$	always	p is true at every state on the path.
$F\,p$	eventually	p is true at some future state on the path.
$p\ U\ q$	until	q is true at some future state, and at every preceding state on path, p is true.

(a) $F\,p$ (b) $p\ U\ q$

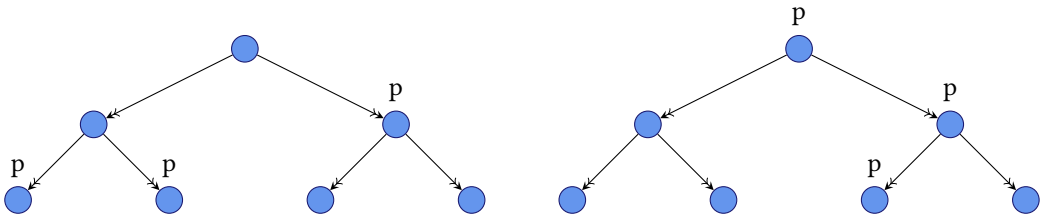
Fig. 6. The state diagram of two example temporal operators in LTL.

2.3 Computational Tree Logic (CTL)

Computation tree logic (CTL) [46] is a branching-time logic, where the future of a state is not determined. To deal with the many possible paths in the future, CTL introduces two types of path qualifiers in Table 4. While "Ap" represents that all possible execution paths from the current state should have the property p, "Ep" represents that there exists at least one path such that p is true. The temporal operator in CTL is composed of a path qualifier and a temporal operator from LTL. For example, $AF\,p$ represents that for all paths from the current state, there must exist one state

Table 4. Path Quantifiers in CTL [85]

Op.	Semantics
$A\ p$	p is true in **all** the paths starting from the current state.
$E\ p$	p is true in **some** path starting in the current state.



(a) *AF p* (b) *EG p*

Fig. 7. The state diagram of two example temporal operators in CTL.

that p is true, as shown in Figure 7(a). Figure 7(b) shows the state diagram of *EG p*, which means that there exists at least one path that p is true for each state along the path.

CTL has the restriction that each LTL operator must be immediately preceded by a path qualifier, e.g., *EG(AF p)*. CTL* is a super-set of both LTL and CTL, and removes this restriction. Although CTL* offers better expressiveness than LTL and CTL, there are still some limitations in CTL*, such as the ability of counting. For example, it cannot express that p should be true for two states before q is true in a path. Researchers have proposed QCTL (quantified CTL) [44] that adds the ability of counting. There are more different variations of temporal logic to solve different problems, but they are beyond the scope of this paper. Section 5 will provide examples of assertions using temporal logic.

3 APPLICATIONS OF ASSERTIONS

While assertions are commonly used for the validation of functional behaviors, assertions are also used for the validation of non-functional requirements. In this section, we explore different use cases of assertions.

3.1 Functional Validation and Debug

Assertions are widely used for functional validation of TLM and RTL models. In this section, we explore seven use cases of assertion-based functional validation.

3.1.1 Control Logic Verification. Components such as buffers, memories, buses and routers on an interconnection network are managed by arbiters and other complex control logic. While directed tests can cover high-level specifications, assertions are used to cover corner cases to ensure all problematic scenarios are detected. For example, ensuring that resources are de-allocated before allocating to another component or checking the correctness of all arbitration schemes (e.g., round-robin, priority, least-recently-used, etc.) can be monitored using assertions [94]. Turumella et al. explored how assertions can be used to verify the control logic of an enterprise-class chip-multi-threaded SPARC™ microprocessor [105].

3.1.2 Finite State Machine Verification. Finite state machines (FSM) are widely used to model systems in different areas such as communication protocols and sequential circuits. The most common errors in finite state machines are deadlock and unreachable states. Deadlocks refer to a state where once entered, no combination of inputs will allow to transition to another state. On the

other hand, unreachable states can never be entered irrespective of the input patterns. Undefined states in FSMs have been known to cause security vulnerabilities as well [109]. Formal verification is one potential solution to capture these errors. However, unbounded and complex properties can make formal verification inefficient. A better way is to capture the intended behavior with multiple assertions. In [70], authors showed how assertions can be used to verify the control block of a UART transmitter that is implemented as an FSM.

3.1.3 Data Integrity Verification. Network-on-chip (NoC) components, bus bridges, direct-memory access controllers and schedulers are some of the main components responsible for data communication between SoC modules. Assertions can be used to check the integrity of data along the complete data route and detect lost or corrupt packets immediately. Assertions written for data integrity verification typically combine end-to-end simulation-based verification and pseudo-random simulation environments that are guided by assertion coverage information [79, 94]. Kakoee et al. presented a NoC verification framework that discussed inter-core as well as intra-core assertion-based verification [63].

3.1.4 Design Interface Verification. When all modules in an SoC are integrated together, verifying their interoperability is critical. While individual modules function as desired, inter-module communication and interface protocol compliance are common failure points [94]. Assertion based protocol monitors are added to catch any malfunctions as soon as possible. A major challenge in writing such assertions is that interface assertions are specific to the architecture of the design and are susceptible to interface description changes. Therefore, the best practice is to code the assertions as the design matures to a level when unit interfaces are reasonably stable. This minimizes the maintenance overhead. Pellauer explored interface assertions in Bluespec System Verilog [87]. Turumella et al.'s work captured inter-module interface assertions in the SPARC^{TM} microprocessor as well [105].

3.1.5 Architectural Coherence Verification. Some of the functionality in an SoC can span multiple SoC components. For example, a cache coherence protocol works across components in the memory hierarchy as well as the NoC IP. Such architectural properties that span across multiple units can be checked using global assertions. Michael et al. [30] discussed a PSL that is ideal for specifying architectural and global assertions. Proper usage of these assertions is useful in finding bugs after all the components are integrated with the SoC.

3.1.6 FPGA Debugging with Assertions. The increase in the transistor count and advanced features found in state-of-the-art FPGAs have made SoC designers choose FPGAs over ASICs. Research by the Garner group revealed that FPGA has a 30:1 edge over ASICs in terms of preference when starting a new design [51]. FPGA SoCs provide flexibility (reconfigurability) but pose additional challenges in test and debug. In fact, debug complexity has become a major bottleneck in pushing FPGA SoCs forward. ABV emerges as a promising solution to this problem. Curreri et al. [41] proposed a high-level synthesis technique to synthesize ANSI-C assertions into an FPGA which enabled verification and debugging of circuits generated from high-level synthesis tools. Their approach can be used to debug while operating the design at full speed in the final system.

3.1.7 Assertion-based Validation of Mixed-signal Designs. While the use cases discussed so far falls in the digital domain, the assertion community has made progress in assertion-based verification in analog and mixed-signal (AMS) domain as well. The tools that were already matured in the digital domain were extended to fit the requirements for verification of AMS systems. One of the major challenges in achieving this goal is the continuous nature of signals in the analog domain. Sammane et al. [10] proposed a method to model an AMS design in terms of a system of recurrence

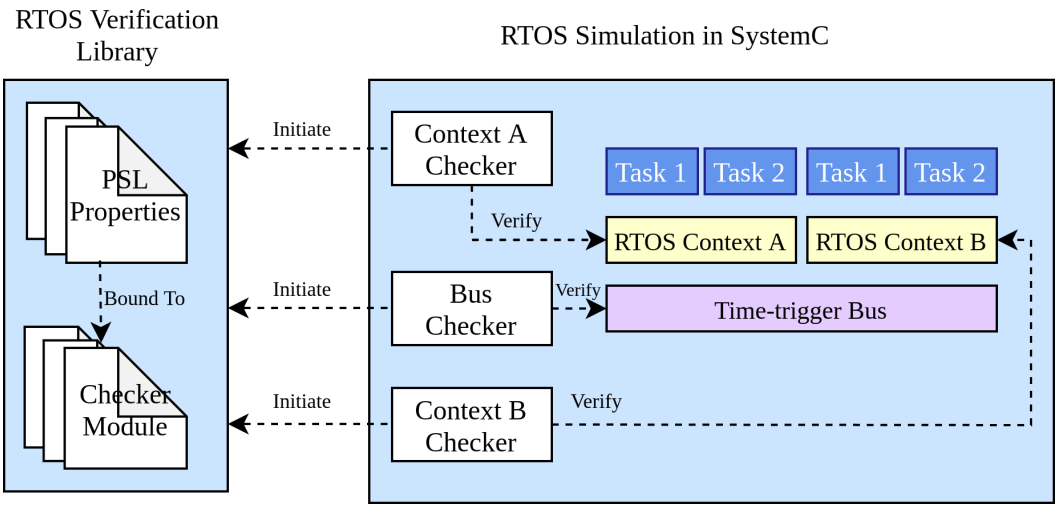


Fig. 8. Application of the RTOS verification library [81]. Checkers are instantiated and connected via ports to test the SystemC model.

equations and apply ABV techniques using the symbolic trace of the equations. An extension to SVA regular expressions that are suitable for continuous-time domain was presented in [58]. This work has contributed to the work of the Accellera committee standardizing Verilog-AMS. Several other methods have also been proposed to address AMS system verification [9, 62, 78, 107]. In this paper, we primarily focus on the assertion-based validation of digital systems.

3.2 Validation of Non-Functional Requirements

We can observe from the use cases in functional validation that assertions define certain properties and check whether they hold or not to guarantee the functional correctness of a design. This analogy can be used to validate other non-functional requirements such as task deadlines, energy and thermal budgets as well as to detect security vulnerabilities.

3.2.1 Detecting timing violations. Embedded systems have different timing constraints depending on the application. Systems that do not run performance-critical applications have soft deadlines where violating a small fraction of deadlines is acceptable. On the other hand, real-time systems have hard deadlines which should not be violated. Assertions can be used to detect violations in both scenarios. Most safety-critical applications run on real-time operating systems (RTOS). Such RTOS-based systems are sensitive to transient faults that can cause scheduling dysfunctions leading to deadline violations. Therefore, detecting faults that can change task execution times/execution flows is critical. Tarillo et al. [102] proposed an assertion based method for fault detection in RTOS. Their work was tested on an assertion-based hardware scheduler that monitors the RTOS in a 32-bit RISC Plasma microprocessor. Mueller et al. [81] discussed how to use PSL-based assertions to check violations in RTOS and how the assertions can be reused and verified at different abstraction levels. Their verification library provided checkers with well-defined interfaces. Figure 8 shows an overview of their verification architecture. Several other works also proposed assertion based solutions to detect timing violations [40, 66, 108].

3.2.2 Detecting energy and thermal constraint violations. The resource-constrained nature of embedded systems introduces more challenges in addition to ensuring performance guarantees. It is not always beneficial to run at the maximum possible frequency to finish a given task early since

it can increase energy consumption. Therefore, the goal is to find optimum points of operation such that the performance-energy trade-off is optimized. In other words, how can a system offer the desired performance without severely degrading battery life. An increase in energy consumption translates to an increase in device temperature as well. As high SoC temperature can have a severe impact, the temperature should be controlled such that it does not exceed a certain threshold. Dynamic Voltage Scaling is widely used for both energy optimization and temperature management. Similar to the usage of assertions in detecting timing violations, assertion-based verification is used to detect energy and temperature budget violations as well. Wang el at. [90] proposed one such method for temperature and energy-constrained scheduling in multitasking systems. In order to define constraints/thresholds, an important part of the design process is to estimate and analyse power consumption at different abstraction levels. Savithri et al. [96] proposed an assertion-based technique for transistor-level dynamic power estimation. In [8], Ahuja et al. proposed an assertion-based electronic system-level power estimation technique for a specific class of design that is modal in nature. In their work, assertions written to verify the reachability of modes is used to generate directed test cases, which are given as inputs to the power estimation framework.

3.2.3 Security vulnerability detection. While there is a vast literature on using assertions for functional verification, there is some initial effort in mitigating security vulnerabilities using assertions. In Section 3.1.2, we discussed how to do functional verification of FSMs using assertions. In the context of security, assertions can be used to ensure that an unspecified transition in an FSM does not allow a user to access a higher privilege level. Similarly, assertions can be used to detect a wide variety of other security vulnerabilities. Witharana et al. [109] outlined several classes of security vulnerabilities that can be detected using assertions such as permissions and privileges, unauthorized resource accesses, illegal states and transitions in FSMs, numeric exceptions, buffer errors, malicious implants and spectre attacks.

4 SURVEY METHODOLOGY

This paper provides a comprehensive survey on four challenging areas of assertion-based verification: automated generation of assertions, test generation for activating assertions, assertion-based pre-silicon verification, and post-silicon debug using synthesized checkers. An overview of our survey methodology is shown in Figure 9.

- **Automated generation of assertions:** Traditional approach for ABV is to write assertions manually. However, writing temporal assertions with high functional coverage is very difficult. Also, any specification change may require rewriting or reevaluating those assertions. Automation helps with these problems. We consider a wide variety of assertions that can capture functional behaviors using Property Specification Language (PSL) [1], SystemVerilog Assertions (SVA) [2] and Open Verification Library (OVL) [3]. Section 5 surveys various assertion generation techniques.
- **Test generation for activation of assertions:** It is a major challenge to ensure that the generated assertions are valid. In other words, we need to make sure that an assertion gets activated when the assertion condition fails during execution. Therefore, the designers need to develop test vectors or for the assertion to fail. Manual development of test vectors can be time consuming and error prone. Section 6 surveys automated test generation techniques for activation assertions.
- **Assertion-based pre-silicon verification:** Assertions are widely used for pre-silicon validation for two reasons: (i) it is beneficial to capture as many bugs as possible in the early stages of the design flow, and (ii) once assertion-based validation is complete, the assertions can be

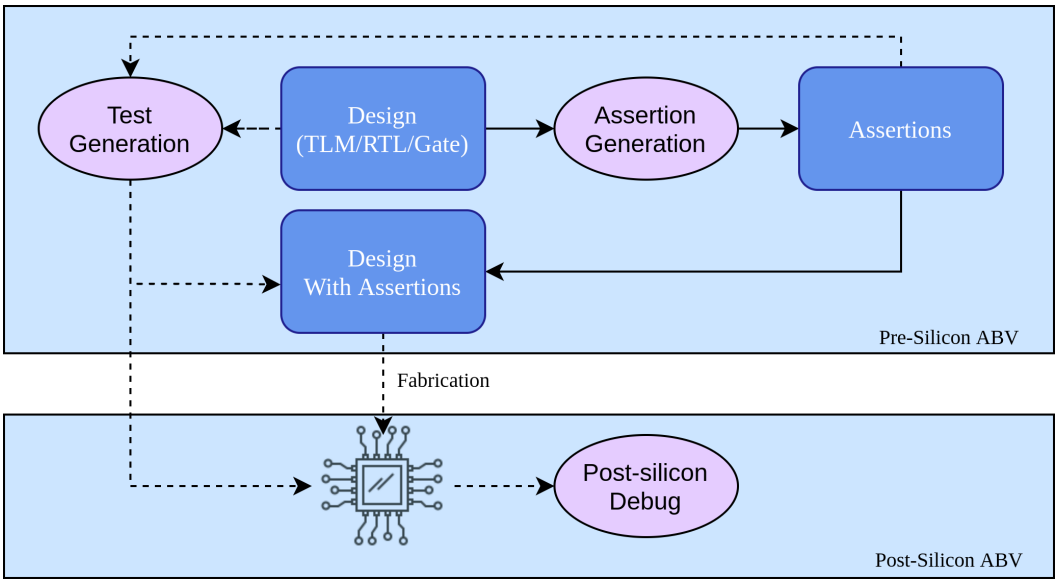


Fig. 9. Overview of assertion-based verification (ABV) survey methodology.

deleted from the design. Section 7 surveys different pre-silicon assertion-based verification techniques.

- **Post-silicon debug using hardware monitors:** It is desirable to map (synthesize) the pre-silicon assertions and retain them in silicon to provide functional coverage during post-silicon debug. Unfortunately, it may not be feasible to synthesize all pre-silicon assertions due to area, power and performance constraints. Section 8 surveys different approaches for assertion-based post-silicon validation and debug.

Due to the increasing importance of assertions, there are a large number of publications in assertion-based validation. This survey mainly includes major publications in the last 20 years that focuses on digital hardware design. In other words, assertions for software verification as well as assertion-based validation of analog and mixed signal systems are beyond the scope of this survey.

5 AUTOMATED GENERATION OF ASSERTIONS

There are mainly two types of approaches to define assertions: language-based and library-based [45]. Language-based approaches have their own syntax for writing assertions. Property Specification Language (PSL) [1] and System Verilog Assertions (SVA) [2] are two popular assertion specification languages. Both PSL and SVA support LTL and CTL extended temporal assertions. Other language examples that support temporal logic include ForSpec [12] and SALT [15]. There are several languages that support RTL assertions. However, SystemC is the standard language for modeling TLM assertions [57, 101]. Figure 16 shows the standard language-based assertions to embed assertions in both TLM and RTL models.

In library-based approaches, assertion support is given to existing languages. One such example is Accellera Open Verification Library (OVL) [3]. OVL supports SystemVerilog, Verilog, VHDL and PSL. OVL is widely used in ARM processor in two ways: (i) assertions in OVL are embedded in the design, and users can enable them, or (ii) a standalone piece of verification IP containing OVL assertions are provided, such as AMBA checkers [53]. The following is an example of a

Verilog-instantiated OVL code and SVA code that check for the case where grant1 and grant2 are not mutually exclusive (example is taken from [54]):

```
// OVL
assert_always mutex (clk, reset_n,
!(grant1 & grant2));

// SVA
property mutex;
   @(posedge clk) disable iff (!reset_n)
   (!(grant1 & grant2));
endproperty
assert property (mutex);
```

The pros and cons of these two approaches are summarized in Table 5. For example, library-based assertions can be used across various designs that have common functional properties but have different languages and implementation details. Although library-based approaches save time in assertion development for common types of assertions, they are not generic enough to cover all possible cases. On the other hand, assertions defined using language-based approaches, such as SVA, are directly embedded into the designs and testbenches. There are two advantages of this scheme. First, the intent of the designer is easier to integrate into the design and documentation in the design phase. It would be hard for a library-based approach to capture during the debug phase. Second, the assertions embedded in the design as well as testbench are allowed to monitor all signals and modules, which is impossible for reused library-based assertions.

Table 5. The comparison between language-based and library-based methods in defining assertions.

Approach	pros	cons
Language-based	Directly embedded into the design and testbench	Language specific
Library-based	Vendor and language independent	Limited scenarios

Example 3: The property $(ack1 \rightarrow \neg ack2)$ for the RTL module in Listing 1 can be converted to an RTL assertion. Since the property needs to be checked concurrently, a concurrent assertion such as *assert property (ack1 |-> !ack2)* can be added to the design. The implication operator($|- >$) in SVA has two expressions. The left expression of the implication is called antecedent and the right expression is called consequent ($antecedent |- > consequent$). If the antecedent expression fails, the implication will be true vacuously without checking the consequent. If the antecedent is true, only the consequent will be evaluated. The assertion will fail if the antecedent was true and consequent was false as shown in Table 6. There are two types of implications in SVA: overlapping($|- >$) and non-overlapping($| =>$). For overlapping implications, if the antecedent is true then the consequent will be evaluated in the same clock cycle. However, for non-overlapping implications, the consequent will be evaluated in the next clock cycle. Since the sample property should hold in the same clock cycle, the assertion is written using overlapping implication. □

Table 6. Assertion States

ack1	ack2	assert property(ack1 |->!ack2)
1	1	Fail
1	0	True
0	1	True (Vacuously)
0	0	True (Vacuously)

Generation of meaningful assertions is a major challenge for design verification. In traditional verification flow, assertions are written manually by verification engineer [54]. However, writing temporal assertions manually is very difficult. Furthermore, the number of assertions must remain reasonable while giving high functional coverage. If there are too many assertions, it will degrade simulation performance. On the other hand, functional coverage can suffer if the number of assertions is small. This problem is further aggravated by frequent design specification and architectural changes. In such cases, the verification engineer may need to rewrite the majority of the assertions or re-validate the existing ones. Thus the automated generation of concise assertions with high coverage is desirable. There are two ways of automated assertion generation: generate assertions by static analysis of design (specification), or dynamic analysis of simulation traces for assertion generation.

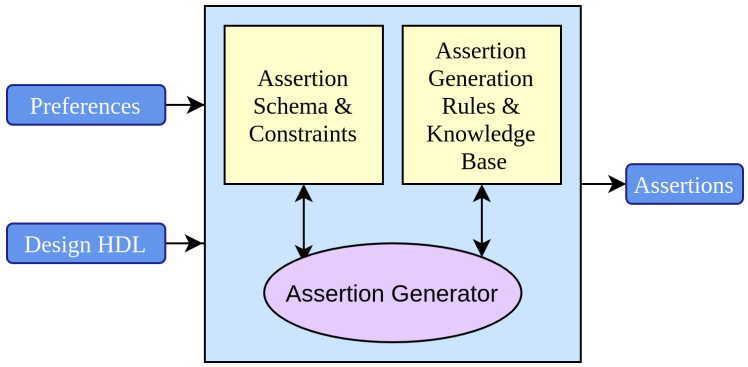


Fig. 10. Template-driven assertion generation using static analysis of specification [59].

5.1 Static Analysis of Specification

Template-driven assertion generation is one of the earlier attempts to automate assertion generation [59]. Figure 10 gives an overview of the process. The generation starts by selecting a schema (template). This schema can be directly used from the schema library without any modification. User can also add new schema or modify existing ones. A process model containing design constructs and constraints is then created by analyzing the schema. As the process models are generated from schema rather than being hard-coded, it increases the applicability of them. This approach has been successfully utilized for the generation of both interface (boundary) assertions and interconnects assertions. Interface assertions applied to input/output ports of the design. These assertions can be used to ensure that each port is exercised and can also check for illegal value combinations. In contrast, interconnect assertions monitor internal ports for value propagation. All the ports that do not generate value propagation are flagged as disconnected. Schema-driven assertion generation requires the design (HDL description) for analysis, making it a white-box approach. This is a limiting factor if the HDL code is not available. Also, there is no formal mechanism to measure the effectiveness of a schema for a particular design. There are various efforts that try to address these challenges. [110] analyzes the syntax and extract properties from a RTL design then generate assertions based on the extracted properties. In contrast, it [59], [110] uses model checking for assertion verification.

5.2 Dynamic Analysis of Simulation Traces

Vasudevan et al. proposed GoldMine for assertion generation of RTL designs [60, 106]. GoldMine generates assertions using static analysis and data mining. It can generate both propositional

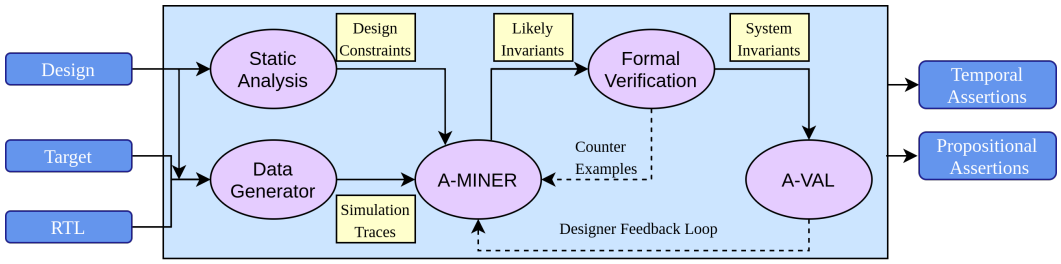


Fig. 11. Assertion generation using both static and dynamic analysis in Goldmine [106].

and temporal assertions. Figure 11 shows the assertion generation procedure. The *data generator* simulates the design under test with random inputs to produce behavioral data of the system. The *static analyzer* extracts design constraints like cone-of-influence, topographical variable ordering, etc. The core assertion miner (*A-MINER*) uses decision tree-based supervised learning [28] to analyze the simulation traces and design constraints. A-Miner finally produces a set of *candidate assertions*. However, these candidate assertions are generated as a result of random input vector, not all possible inputs. The *formal verifier* is used to filter the candidate assertions. Authors have used SMV [77] as their formal verification engine. Finally, the *A-Val* evaluates and ranks the assertions using support and confidence metric. Similar to [59], GoldMine is a white-box approach. GoldMine tries to avoid incorrect assertion generation using formal verification. However, as both simulation and verification are done on the same RTL design and not with a golden model, incorrect RTL implementation can result in incorrect assertions. Furthermore, formal techniques such as model checking are prone to state explosion issue, limiting their applicability on large designs.

Example 4: The Listing 2 shows the functional assertions generated for the RTL design shown in Listing 1. For example, the first property indicates that *ack1* must be false if both *state* and *rec2* are true at the same time. □

Listing 2. Assertions Generated by GoldMine [106]

```
assert property ((state==1 & req2==1) |->(ack1 == 0))
assert property ((req1==1 & state==0) |->(ack1 == 1))
assert property ((req1==0) |-> (ack1==0))
assert property ((req1==1 & req2==0) |->(ack1 == 1))
assert property ((req1==1 & state==0) |->(ack2 == 0))
assert property ((req2==1 & state==1) |->(ack2 == 1))
assert property ((req2==0) |-> (ack2==0))
assert property ((req2==1 & req1==0) |->(ack2 == 1))
```

One of the major limitations of [106] is that it cannot guarantee the correctness of the generated assertions. Liu et al. solved this issue by generating assertions from transaction-level models (TLM) rather than RTL [72]. This assumes that the TLM specification can be used as a golden reference model. Besides, the TLM assertions can be used as templates for RTL assertions. If no assertion can be generated at RTL for each of the TLM assertion, it indicates inconsistency between RTL design and TLM specification. Figure 12 shows the framework for this approach. The first step is to collect simulation traces for data mining. Symbolic simulation is used here unlike concrete simulation as in [106]. Symbolic simulation facilitates generation of fewer, but more comprehensive assertions. For example, consider the assertion generated from the concrete simulation trace [72]: $read\_mem1(100) \rightarrow write\_mem2(100)$. This means reading 100 from mem1 will be followed by writing 100 to mem2. Similarly, using a different address of 200 will generate another assertion: $read\_mem1(200) \rightarrow write\_mem2(200)$. Both of these assertions essentially convey the same

message. Using symbolic simulation instead will produce a much more concise and meaningful assertion: $read\_mem1(A) \rightarrow write\_mem2(A)$. Here A is a symbolic variable denoting the address. After running symbolic simulations, the function call and event traces are collected for data mining. In [106], decision tree-based supervised learning is used for data mining. However, authors have found that sequential pattern mining is more suitable for TLM designs. After this data mining step, we have a set of candidate assertions. An additional post-processing step is employed to convert symbols to their corresponding function calls and events.

Both cycle-accurate and system-level GoldMine generates assertion in bit level [72, 106]. Since assertions are generated for each bit of a word, they are repetitive and can be very large. Such a large number of assertions can be cumbersome for verification engineers. Moreover, the simulation would be extremely slow in the presence of so many assertions. They also make the simulation slower. Furthermore, each of these assertions provides very little functional coverage. To overcome these problems, Liu et al. proposed an approach for word-level feature discovery [71]. According to the discovered word-level targets and features, RTL code is instrumented. Next, the updated RTL is used to generate assertions. This will lead to a reduction of redundant assertions.

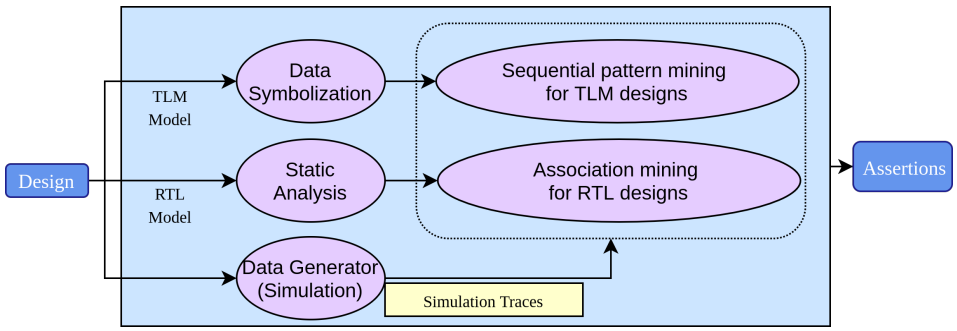


Fig. 12. Automated assertion generation of TLM and RTL designs using GoldMine [72].

The methods described so far do not utilize any coverage related feedback. Thus, an assertion generated by these methods can be redundant and may not improve the functional coverage at all. Furthermore, the method described in [106] uses decision tree which makes the nodes closer to the leaf produce over-constrained (contains a large number of propositions) assertions. Such over-constraining reduces readability and increases simulation time. Sheridan et al. proposed a coverage guided mining approach to circumvent this issue [97]. This method utilizes association rule mining with a greedy set covering and formal verification. Figure 13 provides an overview of this approach. In each iteration, the association rule mining selects assertions that have higher coverage than a threshold and adds them to the candidate assertion set. This ensures that only beneficial assertions (from a functional coverage perspective) are kept as a candidate. After each iteration, the threshold is lowered. Similar to [106], formal verification is carried out to ensure the correctness of candidate assertions. Association rule mining is known to produce more concise rules than decision tree-based algorithms [4]. However, due to their exhaustive nature, they do not scale well with design size. Using coverage threshold-based selection criteria for the assertion helps overcome this scalability issue. According to the authors' experimental results, each assertion generated using this approach covers 6.14 times more input space compared to [106]. Also, the number of propositions per assertion is 2.75 times less.

Recently Danese et al. proposed an approach name A-Team for template-based assertion mining [43]. This method mainly focuses on two problems of the existing assertion mining techniques: i) low flexibility due to pre-defined templates, and ii) redundant, over-constrained assertions.

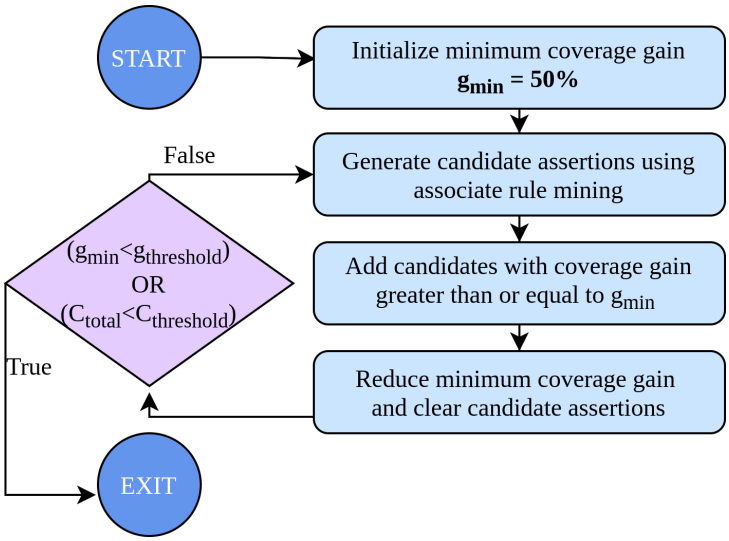


Fig. 13. Coverage guided association mining algorithm [97].

Methodology of A-Team is depicted in Figure 14. It consists of three stages. In the first step, Apriori [5] algorithm is used for mining high frequency atomic propositions. Temporal behavior is not considered in this step. The second step is the mining of LTL assertions. Atomic propositions that were mined during the first step are now composed into LTL assertions. These compositions follow the template given by the user. The final step involves the evaluation of the mined assertions. In this step, each assertion is evaluated against fault coverage. The assertions that do not improve the fault coverage are discarded. Furthermore, a minimization is applied to remove redundant assertions.

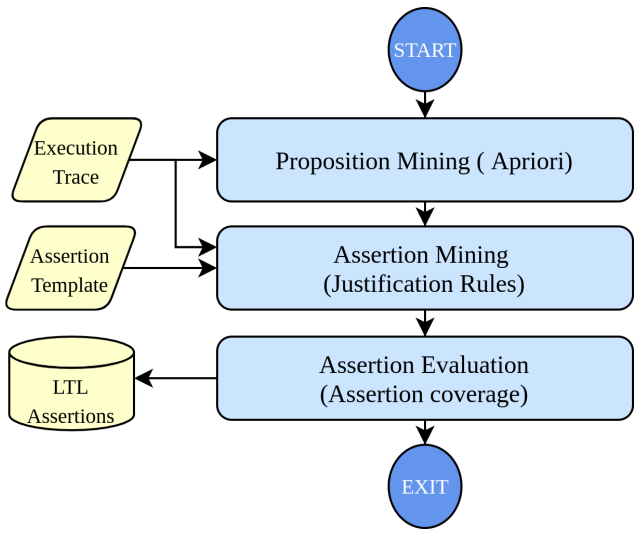


Fig. 14. Template-based assertion mining in A-TEAM [43].

6 TEST GENERATION FOR ACTIVATION OF ASSERTIONS

Since assertions capture complex functional behaviors, it may be hard to activate such assertions using random or constrained-random test patterns. Running too many test patterns implies longer simulation time. A major challenge is how to generate efficient test vectors such that a small set of test vectors can cover all the assertions in the design. The remainder of the section surveys existing test generation techniques for maximizing assertion coverage, including simulation-based approach using random/constrained random tests, directed test generation using formal methods, and a combination of these two approaches.

6.1 Simulation using Random or Constrained-Random Tests

Traditional test generation approaches for assertion activation is simulation-based. Most of simulation-based testing use random or constrained-random test vectors. However, these approaches cannot guarantee that assertions with complex conditions can be activated in a reasonable time. Pal et al. [86] presented a new approach to improve random testing in the RTL level by introducing bias random test generation. In their work [86], the authors propose to use assertions that are only defined for an interface of a module. In other words, they restrict the test generation for assertions with only input/output signals of a module and consider the design as a black box. The results of [86] show that a significant improvement in time for assertion activation compared to using random tests.

Generating test vectors for properties defined by assertions can be utilized to increase the coverage by capturing most of the hard-to-detect cases. Ferro et al [50] use simulation of PSL properties in transaction level. Their framework uses combinatorial testing to select the most suitable stimuli that have the ability to capture all the corner cases. Unlike random testing, combinatorial testing provides a set of test vectors containing all combinations of inputs specified by the designer. Tong et al. [103] introduce a test generator to activate assertions by using compact automata. They perform a state space search in assertion based automata prior to test generation to identify failures and acceptance nodes. This gives a significant advantage in computation time.

6.2 Directed Test Generation using Formal Methods

While random/constrained-random tests can easily cover assertions with simple (easy-to-activate) conditions, they are not suitable for a vast majority of assertions with complex (hard-to-activate) behaviors. Formal methods can enable automated generation of directed tests to address the limitation of random/constrained-random tests [31, 36, 47, 68, 76, 89]. Tong et al. [104] propose a test generation technique for assertion coverage using model checking. The researchers restrict the assertions that only refer to primary inputs. However, this restriction can cause a major drawback for this approach since this approach cannot handle complex assertions which might have complex communications or interactions with internal signals. Earlier approaches of using formal methods require converting the design into a netlist. Recent test generation methods utilize the word-level structure of transition relation which enables the use of Satisfiability Modulo Theories (SMT). Mukherjee et al. [82] highlight the idea of efficient test generation exploiting the word-level structure and SMT solvers.

6.3 Test Generation using Concolic Testing

Concolic testing is a combination of both simulation and symbolic execution. It tries to combine the advantages of both simulation (random tests) and formal methods (directed tests). While it is fast to generate random tests, even millions of random tests may not cover complex assertions (e.g., with rare variables). While formal methods can provide a directed test for a given assertion, formal

methods face the inherent challenge of state space explosion when handling large designs and complex assertions. Concolic testing overcomes the state explosion problem by discovering only one path at a time. While this makes concolic testing scalable, this can lead to a path explosion problem. To overcome the path explosion problem, there are many heuristic solutions. Concolic testing has been successfully used in activating software assertions [69]. Recent efforts utilize concolic testing for test generation using RTL models [6, 7, 73, 75]. Authors in [74] use concolic testing to activate hardware assertions in RTL models. They have converted the assertions to branch statements and used concolic testing with heuristics to activate the assertions. Figure 15 shows an overview of assertion activation using concolic testing that uses a synergistic combination of path simulation and symbolic execution of constraints.

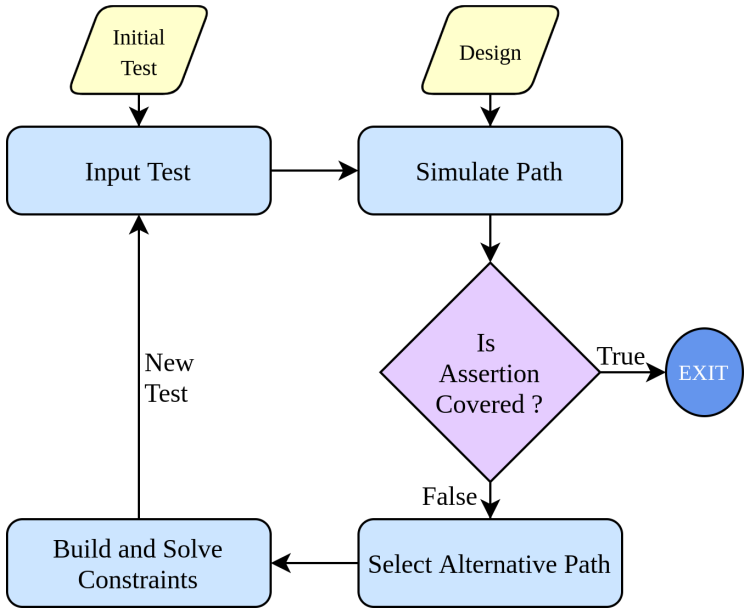


Fig. 15. Assertion activation using concolic testing [74]

7 ASSERTION-BASED PRE-SILICON VERIFICATION

Pre-silicon verification uses assertions to effectively verify a specification using simulation, formal/semi-formal verification as well as hybrid methods. Assertions can be utilized in the design under verification (DUV) to verify functional behavior and improve observability for faster error detection and localization. During pre-silicon validation, assertions are embedded into the design in different stages of the design cycle for various validation purposes, e.g., hardware/software co-design, system-level design and IP-level design. They are implemented either using the same language as the design or as a separate and reusable library as discussed in Section 5. In this section, we first discuss the embedding of assertions at different abstraction levels. Next, we discuss three approaches for assertion-based verification of pre-silicon models.

7.1 Embedding of Assertions at Different Abstraction Levels

A typical SoC design consists of multiple stages, from high-level specification, transaction-level modeling [29], register-transfer level modeling, gate-level netlist, all the way to the post-silicon stage after fabrication. There are three stages that are explored in the literature for assertion-based

validation, i.e., TLM, RTL and post-silicon (shown in Figure 16). From left to right, the model contains more details and is more close to the real design. As a result, the ideal validation should be applied to the post-silicon stage to match the exact behavior of the design. However, there are two major challenges of applying validation in post-silicon stage. First, the validation and debug complexity is drastically higher compared to pre-silicon models. For example, it may take days or weeks to simulate traces of few minutes of silicon execution. Second, the observability in post-silicon is very limited. The available information of the design is limited to the primary input/output and limited traces from debug infrastructures, such as JTAG and trace buffer. On the other hand, the pre-silicon models provide a higher level of abstraction by hiding unnecessary details to validate specific functionalities. The validation in the pre-silicon stage can utilize both simulation and formal methods, and it has the distinct advantage of 100% observability of internal signals. The application of assertions in these stages and their difference are summarized below.

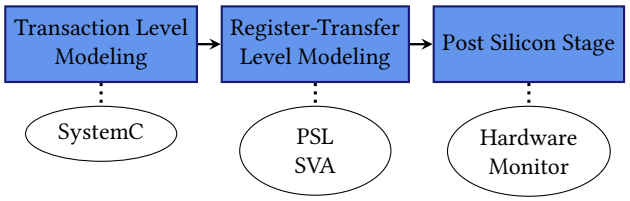


Fig. 16. Assertions are beneficial for both pre-silicon (TLM and RTL) models and post-silicon execution. The TLM model typically utilize SystemC assertions. The RTL model supports both PSL and SVA assertions. The assertions can be used (synthesized) as hardware monitors (checkers) during post-silicon execution.

Both pre-silicon stages in Figure 16 (TLM and RTL) provide some abstraction to the exact behavior in silicon. RTL contains more detailed information compared to TLM, such as interface and timing. As a result, validation in RTL models is closer to silicon. Prior to introduction of validation in TLM models, assertion-based validation in RTL models used to be the de-facto approach. While RTL validation is still viable for medium-size designs, the ever increasing complexity and time-to-market pressure of System-on-Chips (SoCs) requires the validation to be applied to more abstract models. In recent years, TLM validation becomes the standard for system-level design, such as hardware/software co-design, system exploration and verification.

In TLM designs, communication is modeled by channels [55], and transaction requests are handled by interfaces of these channels. There are different types of interfaces, such as bidirectional versus unidirectional, and blocking versus non-blocking. For example, Figure 17 shows the TLM structure of a router design, whose main function is to analyze and distribute the packets received from the master to target slaves [33]. The TLM structure abstracts the communication components as unidirectional non-blocking channels (FIFOs). Compared to RTL models, TLM tries to minimize the amount of information that needs to be processed during simulation, e.g., omitting the details of individual signals. As a result, TLM provides a fast platform to explore architecture alternatives and hardware/software co-design [57].

7.2 Dynamic Verification using Simulation

Assertions can be added either by a designer during the design time or by a verification engineer during the verification stage. These assertions are capable of verifying the functional correctness as well as hard-to-check conditions. Dynamic verification means simulate the design using test patterns and verify coverage of assertions. Figure 18 shows an overview of dynamic verification. Assertions provide two distinct advantages during dynamic verification. First, it provides a mechanism to capture expected functional behaviors. Next, it significantly improves observability. If we try to

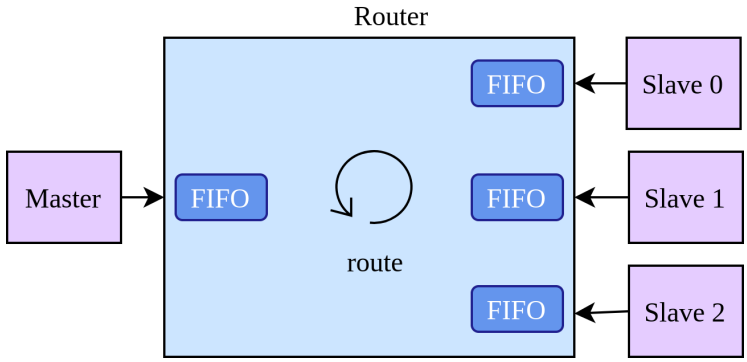


Fig. 17. The TLM structure of the router. [33]

verify a design without assertions, test patterns need to be complex since it can take longer to understand that a specific functionality has been activated. In contrast, coverage of assertions can quickly indicate the activation of the functionality without the need for propagation to the primary (observable) outputs.

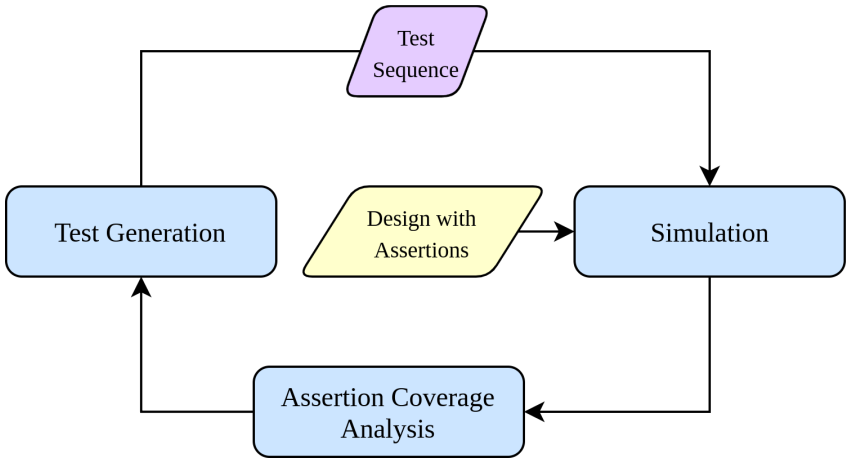


Fig. 18. Overview of dynamic verification using simulation

Assertions can be added in different layers of abstraction including TLM and RTL models. At the RTL level, simulation of design happens according to the rising or falling edge of a clock. Dahan et al. [42] proposed the use of RTL level PSL based assertion verification. They translate the PSL assertions to Verilog assertions and use dynamic simulation to verify the assertions. Authors in [95] automatically transform transaction-level description to RTL assertions. These assertions can be either PSL or VHDL models.

In contrast to the RTL level, TLM models the asynchronous events. TLM level has a higher abstraction hence when writing assertions at TLM level designers do not need to pay attention to all the tedious details like communication between components at the RTL level. Therefore, assertion checking at TLM is more effective than the RTL level. Chen et al. [33] proposed a mechanism to maintain the consistency between TLM and RTL models using assertion based dynamic verification. If a test vector can activate a TLM assertion, the counterpart of test can activate counterparts of

corresponding RTL assertions. Recent efforts [18, 19] reuse RTL assertions in the TLM model to dynamically verify the SystemC TLM specification.

While assertion-based validation of TLM/RTL models is applicable for all designs, there are efforts to explore assertion-based verification specific application scenarios. The paper [105] describes how dynamic ABV is used on the complex enterprise design of a 32-thread SPARC™ CMT microprocessor. It overcame most of the design verification challenges such as exhaustive property checking. The paper [64] uses SVA based assertion verification to debug wireless systems using transaction level modeling. They emphasize that the use of assertions have reduced the network traffic as well. Dynamic ABV can be used to ensure security requirements in a general-purpose processor [17]. In this work, security requirements were harvested using the architectural specification.

7.3 Formal Verification using Model Checking

While simulation is scalable for large designs, it does not provide any mathematical guarantees about the correctness of the design. Formal verification is widely used to prove the functional correctness of hardware models. There are a wide variety of formal verification methods including theorem proving, model checking, satisfiability solving, and equivalence checking. Model (property) checking is most suitable in the context of assertion-based verification since assertions can be viewed as properties.

The utilization of assertions in formal verification has improved the controllability. When the number of required test cases are exponential to verify a functional scenario, it creates low controllability. Low controllability is a disadvantage in dynamic verification. For example, we need 2^{64} test vectors to completely verify a 32-bit adder. Most importantly, it may not be feasible to simulate an exponential number of testcases (with respect to input size) for large designs. In contrast, formal verification of an assertion does not involve any simulation at all. It performs static analysis based on mathematical reasoning to prove that a specific assertion is valid. In case of failure, it will provide a counterexample, so that a verification engineer can debug and either fix the design (if the design is buggy) or the assertion (if the assertion is coded incorrectly). As a result, formal verification of assertions can improve both observability and controllability.

The paper [21] uses real-time assertion verification based on temporal properties represented as PSL assertions. Moreover, they use formal methods to check the properties. One disadvantage in this method is that circuit overhead increases with the number of nested PSL operators. In the paper [105], the authors use formal verification of assertions. Debugging the CMT processor is hard and challenging because of the multi-threaded architecture. Since the hardware is shared, it is really hard to detect a bug or functional error. However, using formal verification of assertions authors have managed to improve the observability of the design and drastically reduce the verification effort.

7.4 Hybrid Verification

While formal verification can provide mathematical guarantees, it has the inherent disadvantage of state space explosion. Therefore, it is not applicable to large designs without suitable abstraction. On the other hand, simulation is scalable but cannot provide correctness guarantees. Clearly, an effective combination of simulation and formal methods would be ideal for assertion-based verification. Rather than using dynamic verification and formal verification separately, the current trend is to utilize both dynamic and static verification of assertions to improve the functional verification of a DUV. In [20], authors use TLM based assertions. First, they use automatic test pattern generation (ATPG) technique to verify the assertions. Then they separately apply a model checker to a set of properties to improve the functional verification.

7.5 Coverage Analysis

In assertion-based verification, high-quality assertions often relate to high coverage. There are three important coverage metrics associated with assertions: how many assertions got activated (*assertion coverage*), how *code coverage* is impacted by the covered assertions, and the coverage of the design functionality achieved by the assertions (*functional coverage*).

Assertion Coverage: As shown in Figure 19, a list of functional behaviors can be derived from the specification. Designers typically write assertions based on these behaviors and embed them in the implementation. Once the implementation is simulated using the generated vectors, the coverage analysis will provide feedback in terms not-activated assertions (more tests needed) or not-covered behaviors (more assertions needed). *Assertion coverage* focuses on how to efficiently activate assertions using test generation techniques. While activation of the assertions guarantees the validity of the assertions, assertion coverage does not imply whether we have checked all the design (functional) behaviors. Therefore, we need to utilize complementary metrics such as code coverage and functional coverage.

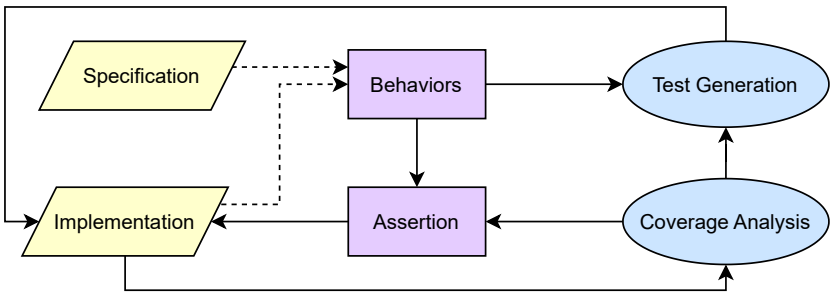


Fig. 19. Coverage Analysis Flow

Code Coverage: *Code coverage* provides an idea of how many lines of code got executed due to the addition of specific assertions. This includes coverage of specific statements, branches, as well as paths during simulation. Code coverage can be used as a metric to identify whether a design has enough assertions. Jayakumar et al. [61] explore the connection between coverage estimation and vacuity detection to compute state coverage. GoldMine[106] uses the conditional coverage which examines an individual path conditions in a RTL design and represent what percentage of conditions covered in a design block. Recent efforts [33, 105] also use the code coverage as a metric to identify whether the number of assertions are enough. Athavale et al. [13] defines a correctness based coverage metric. This work uses a combination of static and dynamic RTL code analysis to identify the covered RTL statements.

Listing 3. Cover and Assert Properties in SVA

```
sequence s1;
  @(posedge clk) a ##1 b;
endsequence

sequence s2;
  @(posedge clk) c;
endsequence

property p;
  s1 |-> s2;
endproperty

// Checker assertion
assert property (p);

// Coverage assertion
cover property (p);
```

Functional Coverage: While code coverage does not guarantee the coverage of functional behaviors, *functional coverage* gives an objective measure of covering design behaviors. Assertions can be used in two ways: checker (to monitor the expected behavior) or coverage (to count if it is activated). Functional coverage of a design can be obtained by the coverage assertion. SVA and PSL languages support both assertion types whereas OVL only support the checker assertion [94]. Illustrative Examples of SVA checker and coverage assertions are shown in Listing 3. There are various efforts in effective utilization of coverage assertions [70, 105]. Knuppel et al. [67] proposes a methodology that considers the degree of incomplete specifications by means of mutation analysis. This feedback can be used to modify the specification. Fedeli et al. [49] performs properties incompleteness evaluation using functional verification. It defines a witness coverage metric for properties using both static and dynamic verification using a fault model targeting functional specification. Specifically, the witness coverage is calculated using input witnesses of the properties extracted from the golden design. Although we have discussed coverage analysis in the context of pre-silicon assertion-based validation, some of these concepts are also useful for coverage analysis of post-silicon checkers.

Example 5: Assertions can be activated both vacuously and non-vacuously. Table 7 shows the test vectors for each case of the assertion (*assert property(ack1 |-> !ack2)*) in the Listing 1. In this example, we cannot generate a test pattern (shown as NULL) to activate the scenario of both *ack*1 and *ack*2 as true. We should be able to generate a test to activate the scenarior, if we insert a vulnerability to make both *ack*1 and *ack*2 as true. We can insert To non-vacuously activate the assertion *ack*1 and *ack*2 should be 1 and 0, respectively. The test pattern <rst=1, req1=1, req2=1> is able to activate the assertion non-vacuously. □

Table 7. Test Vectors

ack1	ack2	assert property(ack1 |->!ack2)	Test Vector
1	1	Fail	NULL
1	0	True	<rst=1, req1=1, req2=1>
0	1	True (Vacuously)	<rst=1, req1=0, req2=1>
0	0	True (Vacuously)	<rst=1, req1=0, req2=0>

8 POST-SILICON DEBUG USING HARDWARE MONITORS

Although most assertions are defined in the pre-silicon stage, they can also be synthesized to the post-silicon stage in the form of coverage monitors. Despite the extensive validation efforts in the pre-silicon stage, assertions are still critical in the post-silicon stage for several reasons. First, the behaviors of silicon and the simulation/emulation are different due to asynchronous interfaces, potential manufacture errors, soft errors during run-time, etc. Next, pre-silicon validation may not be able to capture all possible scenarios. Critical properties still need to be enforced before any catastrophe happens. Finally, monitors can improve observability during post-silicon debug. Monitors provide a more straightforward and accurate way to test a property during runtime compared to other design-for-debug features, such as trace buffer.

Example 6: During synthesis, the assertion shown in the sample RTL module in Listing 1 can be converted to a hardware monitor with a trigger. The trigger for the assertion *assert property(ack1 |-> !ack2)* will be ($ack1 \ \ \& \ \ ack2$). The trigger will be true when both $ack1$ and $ack2$ are 1 at the same clock cycle. This is similar to the behaviour of assertion failure. As shown in Listing 4, the flag value will be either 1 or 0 depending on the trigger condition. During post-silicon debug, only input and output ports will be observable. One way to observe the $flag$ wire is mapping it to an output port. However this method is not practical for complex designs. One practical way to observe the internal states is using an Embedded Trace Buffer (ETB) with suitable signal selection [14, 80, 91–93]. As shown in the Listing 4, when the flag is 1 required states are sent to the ETB via the *ETB_bus* wire. □

Listing 4. Assertion Checker for Post-Silicon Debug

```
//assert property(ack1 |-> !ack2)
wire flag;
wire [31:0]ETB_bus;
assign flag = (ack1 & ack2) ? 1 : 0;
assign ETB_bus = flag ? {ack1, ack2} : 0;
```

In spite of the usefulness of hardware monitors, we cannot add an arbitrary number of monitors compared to the pre-silicon stage, since each monitor will introduce extra area, power and energy overhead. To balance the trade-off between observability and design overhead, Farahmandi et al. [48] combined both advantages of monitors and trace buffer. Monitors whose results can be easily analyzed from the trace buffer are omitted. Only the hard-to-detect ones are synthesized. They also proposed an observability-aware trace signal selection algorithm to cover as many monitors as possible by the trace buffer. In this way, high observability is achieved with low hardware overhead.

Hardware checkers are assertions that are incorporated into the design for runtime monitoring. We have seen that assertions are used to detect deviation from expected behavior. They can be converted into permanent circuitry which is then used for online monitoring or performing self-tests. These checkers are written in synthesizable hardware description languages, as opposed to PSL or SVA. Some of the usages of such assertion checker are depicted in Figure 20. Assertion checkers can be used in the verification stage when the design is under emulation. These checkers are temporary and only present during verification. Assertion checkers can also be used in post-silicon debug as a debug interface as well as for self testing. These checkers are synthesized and added to the design, which remains there during the lifespan of the device. The remainder of this section outlines two components of post-silicon debug using assertions: checker creation and checker utilization.

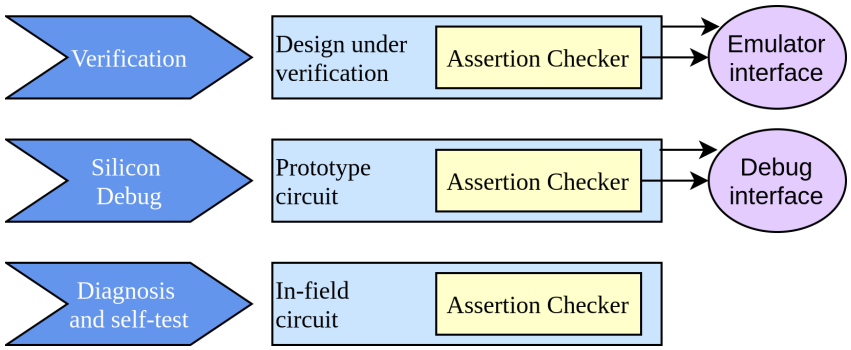


Fig. 20. Usage scenarios for hardware assertion checkers [22].

8.1 Creation of Assertion Checkers (Hardware Monitors)

Checkers can be included in both the pre-silicon and post-silicon phase. In pre-silicon, there are mainly two types of assertion checkers: built-in checkers and post-processing checkers. Built-in checkers are included in the RTL model whereas post-processing checkers evaluate trace files after the simulation [65]. Built-in checkers are always active and monitor the behaviour in every simulated cycle which simplify the debugging and identify the bugs effectively. One more advantage of built-in checkers is that a checker can be used to identify different deviations irrespective of the intended use. However, these checkers slow down the simulation speed. For the majority of the built-in checkers, the impact of the speed of simulation was negligible but for complex checkers impact was high. As a solution post-processing checkers were introduced [65] for complex assertions where the checkers were separated from the model. Post processing checkers will use the dumped trace file of the simulation and identify whether any deviation from the intended design has occurred.

In post-silicon verification and when adding checkers to the circuit, one key challenge is to determine which assertions should be added to the design as hardware checkers. The simplest approach is to include all of them. While it maximizes the observability and run-time debug capability, the power, performance and area concerns are the limiting factors. Designers must strike a balance between these trade-offs. We will look into the existing checker selection methodologies in this section.

One approach for checker selection is static synthesis using different ranking algorithms. The work [48] presents a framework that can be used to reduce the number of hardware checkers in post-silicon validation while using the debug infrastructure. [48] uses the on-chip trace buffer and rank the assertions based on the difficulty in detecting the assertions. Then assertions that are hard-to-detect are synthesized in the design ignoring the easy-to-detect assertions. Similarly in [98–100], the authors rank the assertions based on the ability to detect bit flips and synthesize highly ranked assertions for post-silicon validation to identify electrical bugs. The ranking mechanism checks whether the destination signals of assertions point to flip-flops and then rank those assertions as a high priority.

Another approach is the dynamic synthesis of checkers using FPGA [56]. The work [56] introduces Time-Multiplexed Assertion Checking (TMAC) where the assertion checkers are included in a re-configurable embedded block (FPGA) in a time-multiplexed manner. This gives the capability to add a large number of checkers with a low area overhead.

There are different methodologies to improve the post-silicon debug without changing the number of checkers included. Works [83], [84] present that efficient post-silicon debugging can

be achieved by clustering the assertion checkers. When the assertions are clustered, it is easy to select and control assertions in debug mode whereas individual assertion control and integration will require more energy. The overhead of incorporating checkers in the design can be reduced by selecting an efficient synthesis mechanism. The process of converting assertions into hardware checkers involves transforming the PSL statements into RTL models suitable for use in the circuit under verification. The paper [23] present MBAC hardware checker generator that utilize the limited resources and produce assertion circuits which are resource-efficient, synthesizable and behaviorally correct. In works [24, 25], automata-based assertion checker synthesis is used to generate more resource efficient checkers.

The book [27] presents several debugging improvements introduced for checkers. These enhancements are reporting signal dependencies, monitoring activities, signalling assertion completion, automatically create counters on assert statements and hardware assertion threading. In [26, 37], debugging enhancements such as increasing the observability and the coverage information given by checkers were introduced. These enhancements were achieved by extracting the failures using dependency graphs, monitoring activities of automata-based assertion checkers and monitoring assertion completion. In [26], assertion threading was used to closely monitor which start condition has caused an error. Assertion threading can separate different parallel activities and identify which activity has caused an error from chains of events.

8.2 Utilization of Assertion Checkers (Hardware Monitors)

Checkers are mostly used to identify functional correctness in a design. Functional errors occur when the actual design implementation has some deviations from specification. However prior work shows that checkers can also be used to identify security vulnerabilities, electrical bugs and thermal effects in a circuit.

Hardware Trojan is a major security vulnerability in SoC designs. Hardware Trojans are additional hardware maliciously implanted in a circuit to gain unauthorized access or privileges over an original design. [11] use hardware checkers to identify Trojans. They introduce assertion checkers in programmable logic block to identify Trojans inside a circuit during run-time. In [17], researchers combine hardware checkers with code coverage in order to efficiently detect malicious implants and rare-event triggers.

Electrical bugs and thermal effects are not easy to detect in pre-silicon validation because of the difficulties in accurately modelling them. Electrical bugs are most commonly shown as bitflips in the post-silicon phase [99]. Therefore in [98–100], researchers have used hardware checkers to identify bit-flips in flip-flops so that the electrical errors can be minimized. They prioritize checkers that are able to detect bitflips in flip-flops by examining whether the designated signal points to a flip-flop.

9 CONCLUSION AND FUTURE DIRECTIONS

Assertion-based verification is one of the promising verification techniques used in the industry for hardware designs. Using assertions will improve the controllability and observability which will help faster localization of errors and reduce debug time. There are different abstraction levels, verification levels and verification techniques associated with assertions. This paper surveys assertion-based verification across different abstract levels and verification levels. Moreover, this paper systematically reviews the most recent progress in assertion-based hardware verification. It includes assertion generation techniques, test generation for covering these assertions, hardware checkers and different verification schemes.

Although assertions have been widely used in pre-silicon validation, how to fully utilize existing design-for-debug infrastructure to reduce the number of assertions for post-silicon debug, remains

an open research problem. Future research efforts, especially on IoT devices, need to take all parameters into consideration, including area, power and performance.

The vast majority of existing work deals with assertion-based verification of functional behaviors. As a result, these assertions are written to hold properties that are always true. A recent work [109] utilizes assertions to detect security vulnerabilities and introduce the notion of SoC security assertions. The security assertions are introduced for eight classes of vulnerabilities. Their results show that normal assertions are not good enough to detect all the vulnerabilities and there is a need for dedicated security assertions. Future research in assertion-based verification can focus on defining and utilizing assertions for functional behaviors as well as non-functional requirements such as security, energy and thermal constraints.

10 ACKNOWLEDGMENTS

This work was partially supported by grants from National Science Foundation (CCF-1908131) and Semiconductor Research Corporation (2020-CT-2934).

REFERENCES

[1] 2010. 1850-2010 - IEEE Standard for Property Specification Language (PSL).
[2] 2012. 1800-2012 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language.
[3] 2020. Open Verification Language. https://www.accellera.org/downloads/standards/ovl. Accessed: 2020-02-11.
[4] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. In *Acm sigmod record*, Vol. 22. ACM, 207–216.
[5] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, Vol. 1215. 487–499.
[6] Alif Ahmed, Farimah Farahmandi, and Prabhat Mishra. 2018. Directed test generation using concolic testing on RTL models. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Dresden, Germany, 1538–1543.
[7] Alif Ahmed and Prabhat Mishra. 2017. QUEBS: Qualifying event based search in concolic testing for validation of RTL models. In *IEEE International Conference on Computer Design (ICCD)*. 185–192.
[8] Sumit Ahuja, Deepak A Mathaikutty, Sandeep Shukla, and Ajit Dingankar. 2007. Assertion-based modal power estimation. In *2007 Eighth International Workshop on Microprocessor Test and Verification*. IEEE, 3–7.
[9] Antara Ain, Antonio Anastasio Bruto da Costa, and Pallab Dasgupta. 2016. Feature indented assertions for analog and mixed-signal validation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 11 (2016), 1928–1941.
[10] Ghiath Al Sammane, Mohamed H Zaki, Zhi Jie Dong, and Sofiene Tahar. 2007. Towards Assertion Based Verification of Analog and Mixed Signal Designs Using PSL.. In *FDL*. 293–298.
[11] Uthman Alsaiari, Fayez Gebali, and Mostafa Abd-El-Barr. 2017. Programmable assertion checkers for hardware Trojan detection. In *2017 1st Conference on PhD Research in Microelectronics and Electronics Latin America (PRIME-LA)*. IEEE, 1–4.
[12] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Vardi, and Yael Zbar. 2002. The ForSpec temporal logic: A new temporal property-specification language. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 296–311.
[13] Viraj Athavale, Sai Ma, Samuel Hertz, and Shobha Vasudevan. 2014. Code coverage of assertions using RTL source code analysis. In *Proceedings of the 51st Annual Design Automation Conference*. 1–6.
[14] Kanad Basu and Prabhat Mishra. 2012. RATS: Restoration-aware trace signal selection for post-silicon validation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21, 4 (2012), 605–613.
[15] Andreas Bauer and Martin Leucker. 2011. The theory and practice of SALT. In *NASA Formal Methods Symposium*. Springer, 13–40.
[16] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. 1983. The temporal logic of branching time. *Acta informatica* 20, 3 (1983), 207–226.
[17] Michael Bilzor, Ted Huffmire, Cynthia Irvine, and Tim Levin. 2012. Evaluating security requirements in a general-purpose processor by combining assertion checkers with code coverage. In *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*. IEEE, 49–54.

[18] Nicola Bombieri, Riccardo Filippozzi, Graziano Pravadelli, and Francesco Stefanni. 2015. RTL property abstraction for TLM assertion-based verification. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 85–90.

[19] Nicola Bombieri, Franco Fummi, Valerio Guarnieri, Graziano Pravadelli, Francesco Stefanni, Tara Ghasempouri, Michele Lora, Giovanni Auditore, and Mirella Negro Marcigaglia. 2014. On the reuse of RTL assertions in SystemC TLM verification. In *Test Workshop-LATW, 2014 15th Latin American*. IEEE, 1–6.

[20] Nicola Bombieri, Franco Fummi, Graziano Pravadelli, and Andrea Fedeli. 2007. Hybrid, incremental assertion-based verification for TLM design flows. *IEEE Design & Test of Computers* 24, 2 (2007).

[21] Dominique Borrione, Miao Liu, Katell Morin-Allory, Pierre Ostier, and Laurent Fesquet. 2005. On-line assertion-based verification with proven correct monitors. In *Information and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on*. IEEE, 125–143.

[22] Marc Boule, Jean-Samuel Chenard, and Zeljko Zilic. 2007. Assertion checkers in verification, silicon debug and in-field diagnosis. In *Quality Electronic Design, 2007. ISQED'07. 8th International Symposium on*. IEEE, 613–620.

[23] Marc Boule and Zeljko Zilic. 2005. Incorporating efficient assertion checkers into hardware emulation. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*. IEEE, 221–228.

[24] Marc Boulé and Zeljko Zilic. 2006. Efficient automata-based assertion-checker synthesis of PSL properties. In *High-Level Design Validation and Test Workshop, 2006. Eleventh Annual IEEE International*. IEEE, 69–76.

[25] Marc Boulé and Zeljko Zilic. 2007. Efficient automata-based assertion-checker synthesis of SEREs for hardware emulation. In *2007 Asia and South Pacific Design Automation Conference*. IEEE, 324–329.

[26] Marc Boulé and Zeljko Zilic. 2008. Assertion checkers-enablers of quality design. In *Microsystems and Nanoelectronics Research Conference, 2008. MNRC 2008. 1st*. IEEE, 97–100.

[27] Marc Boulé and Zeljko Zilic. 2008. *Generating hardware assertion checkers*. Springer.

[28] Leonard A Breslow and David W Aha. 1997. Simplifying decision trees: A survey. *The Knowledge Engineering Review* 12, 1 (1997), 1–40.

[29] Lukai Cai and Daniel Gajski. 2003. Transaction Level Modeling: An Overview. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. Association for Computing Machinery, New York, NY, USA, 19–24. https://doi.org/10.1145/944645.944651

[30] C Michael Chang and Harry D Foster. 2003. Property specification: the key to an assertion-based verification platform. (2003).

[31] Mingsong Chen and Prabhat Mishra. 2010. Functional test generation using efficient property clustering and learning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29, 3 (2010), 396–404.

[32] Mingsong Chen and Prabhat Mishra. 2013. Assertion-based functional consistency checking between TLM and RTL models. In *International Conference on VLSI Design*. 320–325.

[33] Mingsong Chen and Prabhat Mishra. 2013. Assertion-based functional consistency checking between TLM and RTL models. In *VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), 2013 26th International Conference on*. IEEE, 320–325.

[34] Mingsong Chen, Prabhat Mishra, and Dhrubajyoti Kalita. 2007. Towards RTL test generation from SystemC TLM specifications. In *IEEE International High Level Design Validation and Test Workshop*. 91–96.

[35] Mingsong Chen, Prabhat Mishra, and Dhrubajyoti Kalita. 2012. Automatic RTL test generation from SystemC TLM specifications. *ACM Transactions on Embedded Computing Systems (TECS)* 11, 2 (2012), 1–25.

[36] Mingsong Chen, Xiaoke Qin, Heon-Mo Koo, and Prabhat Mishra. 2012. *System-level validation: high-level modeling and directed test generation techniques*. Springer.

[37] Jean-Samuel Chenard, Stephan Bourduas, Nathaniel Azuelos, Marc Boulé, and Zeljko Zilic. 2007. Hardware assertion checkers in on-line detection of faults in a hierarchical-ring network-on-chip. In *Workshop on Diagnostic Services in Network-on-Chips*. 371–375.

[38] Edmund M Clarke. [n.d.]. Lecture on propositional logic. https://www.cs.cmu.edu/ emc/15414-f12/lecture/propositional-logic.pdf. Accessed: 2018-04-20.

[39] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2000. *Model Checking*. MIT Press, Cambridge, MA, USA.

[40] Silviu S Craciunas and Ramon Serna Oliver. 2016. Combined task-and network-level scheduling for distributed time-triggered systems. *Real-Time Systems* 52, 2 (2016), 161–200.

[41] John Curreri, Greg Stitt, and Alan D George. 2010. High-level synthesis techniques for in-circuit assertion-based verification. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 1–8.

[42] Anat Dahan, Daniel Geist, Leonid Gluhovsky, Dmitry Pidan, Gil Shapir, Yaron Wolfsthal, Lyes Benalycherif, R Kamidem, and Younes Lahbib. 2005. Combining system level modeling with assertion based verification. In *Quality of Electronic Design, 2005. ISQED 2005. Sixth International Symposium on*. IEEE, 310–315.

[43] Alessandro Danese, Nicolò Dalla Riva, and Graziano Pravadelli. 2017. A-TEAM: Automatic template-based assertion miner. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 1–6.

[44] Amélie David, Francois Laroussinie, and Nicolas Markey. 2016. On the Expressiveness of QCTL. In *27th International Conference on Concurrency Theory (CONCUR 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 59)*, Josée Desharnais and Radha Jagadeesan (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 28:1–28:15. https://doi.org/10.4230/LIPIcs.CONCUR.2016.28

[45] Giuseppe Di Guglielmo, Luigi Di Guglielmo, Andreas Foltinek, Masahiro Fujita, Franco Fummi, Cristina Marconcini, and Graziano Pravadelli. 2013. On the integration of model-driven design and dynamic assertion-based verification for embedded software. *Journal of Systems and Software* 86, 8 (2013).

[46] E.Allen Emerson and Edmund M. Clarke. 1982. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming* 2, 3 (1982), 241 – 266. https://doi.org/10.1016/0167-6423(83)90017-5

[47] Farimah Farahmandi and Prabhat Mishra. 2018. Automated test generation for debugging multiple bugs in arithmetic circuits. *IEEE Trans. Comput.* 68, 2 (2018), 182–197.

[48] Farimah Farahmandi, Ronny Morad, Avi Ziv, Ziv Nevo, and Prabhat Mishra. 2017. Cost-effective analysis of post-silicon functional coverage events. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 392–397.

[49] Andrea Fedeli, Franco Fummi, and Graziano Pravadelli. 2007. Properties incompleteness evaluation by functional verification. *IEEE Trans. Comput.* 56, 4 (2007), 528–544.

[50] Luca Ferro, Laurence Pierre, Yves Ledru, and Lydie du Bousquet. 2008. Generation of test programs for the assertion-based verification of TLM models. In *Design and Test Workshop, 2008. IDT 2008. 3rd International*. IEEE, 237–242.

[51] Harry Foster. [n.d.]. Improving FPGA Debugging with Assertions. https://verificationacademy.com/news/improving-fpga-debugging-assertions. Accessed: 2020-02-11.

[52] Harry Foster. [n.d.]. Wilson Research Group Functional Verification Study 2020.

[53] Harry Foster, Kenneth Larsen, and Mike Turpin. 2006. Introduction to the new accellera open verification library. In *DVCon'06: Proceedings of the Design and Verification Conference and exhibition*. Citeseer.

[54] Harry D Foster, Adam C Krolnik, and David J Lacey. 2004. *Assertion-based design*. Springer Science & Business Media.

[55] Daniel D. Gajski, Jianwen Zhu, Rainer Dümer, Andreas Gerstlauer, and Shuqing Zhao. 2000. *SPECC: Specification Language and Methodology*. Springer US, Boston, MA. https://doi.org/10.1007/978-1-4615-4515-6

[56] Ming Gao and Kwang-Ting Cheng. 2010. A case study of time-multiplexed assertion checking for post-silicon debugging. In *High Level Design Validation and Test Workshop (HLDVT), 2010 IEEE International*. IEEE, 90–96.

[57] Frank Ghenassia (Ed.). 2005. *Transaction Level Modeling with SystemC*. Springer US. https://doi.org/10.1007/b137175

[58] John Havlicek and Scott Little. 2011. Realtime regular expressions for analog and mixed-signal assertions. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 155–162.

[59] Amir Hekmatpour and Azadeh Salehi. 2005. Block-based schema-driven assertion generation for functional verification. In *Test Symposium, 2005. Proceedings. 14th Asian*. IEEE, 34–39.

[60] Samuel Hertz, David Sheridan, and Shobha Vasudevan. 2013. Mining hardware assertions with guidance from static analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 6 (2013), 952–965.

[61] Nikhil Jayakumar, Mitra Purandare, and Fabio Somenzi. 2003. Do's and don'ts of CTL state coverage estimation. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451)*. IEEE, 292–295.

[62] Alexander Jesser, Stefan Laemmermann, A Pacholik, R Weiss, Juergen Ruf, W Fengler, L Hedrich, T Kropf, and Wolfgang Rosenstiel. 2007. Analog Simulation Meets Digital Verification–A Formal Assertion Approach for Mixed-Signal Verification. In *SASIMI*, Vol. 7. 507–514.

[63] Mohammad Reza Kakoee, Mohammad Hossein Neishaburi, Masoud Daneshtalab, Saeed Safari, and Zainalabedin Navabi. 2007. On-chip verification of nocs using assertion processors. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*. IEEE, 535–538.

[64] Vivek N Kallankara, MH Neishaburi, Katarzyna Radecka, and Zeljko Zilic. 2010. Using assertions for wireless system monitoring and debugging. In *NEWCAS Conference (NEWCAS), 2010 8th IEEE International*. IEEE, 401–404.

[65] Michael Katrowitz and Lisa M Noack. 1996. I'm done simulating; now what? Verification coverage analysis and correctness checking of the DEC chip 21164 Alpha microprocessor. In *Proceedings of the 33rd annual Design Automation Conference*. ACM, 325–330.

[66] Steve Kerrison, David May, and Kerstin Eder. 2016. A Benes Based NoC switching architecture for mixed criticality embedded systems. In *2016 IEEE 10th international symposium on embedded multicore/many-core systems-on-chip (MCSOC)*. IEEE, 125–132.

[67] Alexander Knüppel, Leon Schaer, and Ina Schaefer. 2021. How much Specification is Enough? Mutation Analysis for Software Contracts. In *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*. IEEE, 42–53.

[68] Heon-Mo Koo and Prabhat Mishra. 2009. Functional test generation using design and property decomposition techniques. *ACM Transactions on Embedded Computing Systems (TECS)* 8, 4 (2009), 1–33.
[69] Bogdan Korel and Ali M Al-Yami. 1996. Assertion-oriented automated test data generation. In *Proceedings of the 18th international conference on Software engineering*. IEEE Computer Society, 71–80.
[70] Yangyang Li, Wuchen Wu, Ligang Hou, and Hao Cheng. 2009. A study on the assertion-based verification of digital IC. In *2009 Second International Conference on Information and Computing Science*, Vol. 2. IEEE, 25–28.
[71] Lingyi Liu, Chen-Hsuan Lin, and Shobha Vasudevan. 2012. Word level feature discovery to enhance quality of assertion mining. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*. IEEE, 210–217.
[72] Lingyi Liu, David Sheridan, Viraj Athavale, and Shobha Vasudevan. 2011. Automatic generation of assertions from system level design using data mining. In *Proceedings of the Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign*. IEEE Computer Society, 191–200.
[73] Yangdi Lyu, Alif Ahmed, and Prabhat Mishra. 2019. Automated activation of multiple targets in RTL models using concolic testing. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Florence, Italy, 354–359.
[74] Yangdi Lyu and Prabhat Mishra. 2020. Automated Test Generation for Activation of Assertions in RTL Models. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 223–228.
[75] Yangdi Lyu and Prabhat Mishra. 2020. Scalable concolic testing of RTL models. *IEEE Trans. Comput.* (2020).
[76] Yangdi Lyu, Xiaoke Qin, Mingsong Chen, and Prabhat Mishra. 2018. Directed test generation for validation of cache coherence protocols. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 1 (2018), 163–176.
[77] Kenneth L McMillan. 1993. The SMV system. In *Symbolic Model Checking*. Springer, 61–85.
[78] Ashok B Mehta. 2018. Analog/mixed signal (ams) verification. In *ASIC/SoC Functional Design Verification*. Springer, 255–271.
[79] Ashok B Mehta. 2020. System verilog assertions. In *System Verilog Assertions and Functional Coverage*. Springer, 11–31.
[80] Prabhat Mishra and Farimah Farahmandi. 2019. *Post-Silicon Validation and Debug*. Springer.
[81] Wolfgang Mueller, Marcio F da S Oliveira, Henning Zabel, and Markus Becker. 2010. Verification of real-time properties for hardware-dependent software. In *2010 IEEE International High Level Design Validation and Test Workshop (HLDVT)*. IEEE, 154–159.
[82] R. Mukherjee, D. Kroening, and T. Melham. 2015. Hardware Verification Using Software Analyzers. In *2015 IEEE Computer Society Annual Symposium on VLSI*. IEEE, Montpellier, France, 7–12. https://doi.org/10.1109/ISVLSI.2015.107
[83] Mohammad Hossein Neishaburi and Zeljko Zilic. 2010. Enabling efficient post-silicon debug by clustering of hardware-assertions. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*. IEEE, 985–988.
[84] Mohammad Hossein Neishaburi and Zeljko Zilic. 2012. An infrastructure for debug using clusters of assertion-checkers. *Microelectronics Reliability* 52, 11 (2012), 2781–2798.
[85] Antti Pakonen, Cheng Pang, Igor Buzhinsky, and Valeriy Vyatkin. 2016. User-friendly formal specification languages-conclusions drawn from industrial experience on model checking. In *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*. IEEE, 1–8.
[86] Bhaskar Pal, Ansuman Banerjee, Arnab Sinha, and Pallab Dasgupta. 2008. Accelerating assertion coverage with adaptive testbenches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 5 (2008), 967–972.
[87] Michael Pellauer, Mieszko Lis, Don Baltus, and Rishiyur Nikhil. 2005. Synthesis of synchronous assertions with guarded atomic actions. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE'05.* IEEE, 15–24.
[88] A. Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 46–57. https://doi.org/10.1109/SFCS.1977.32
[89] Xiaoke Qin and Prabhat Mishra. 2012. Directed test generation for validation of multicore architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 17, 3 (2012), 1–21.
[90] Xiaoke Qin, Weixun Wang, and Prabhat Mishra. 2012. TCEC: Temperature and energy-constrained scheduling in real-time multitasking systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 8 (2012), 1159–1168.
[91] Kamran Rahmani and Prabhat Mishra. 2017. Feature-based signal selection for post-silicon debug using machine learning. *IEEE Transactions on Emerging Topics in Computing* 8, 4 (2017), 907–915.
[92] Kamran Rahmani, Sudhi Proch, and Prabhat Mishra. 2015. Efficient selection of trace and scan signals for post-silicon debug. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 1 (2015), 313–323.
[93] Kamran Rahmani, Sandip Ray, and Prabhat Mishra. 2016. Postsilicon trace signal selection using machine learning techniques. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 2 (2016), 570–580.

[94] Zhihong Ren and Hussain Al-Asaad. 2016. Overview of Assertion-Based Verification and its Applications. In *Int'l Conf. Embedded Systems, Cyber-physical Systems, & Applications*.
[95] Aurélien Ribon, Bertrand Le Gal, Christophe Jégo, and Dominique Dallet. 2011. Assertion support in high-level synthesis design flow. In *Specification and Design Languages (FDL), 2011 Forum on*. IEEE, 1–8.
[96] S Savithri, R Venkatesan, and S Bhaskar. 2000. An assertion based technique for transistor level dynamic power estimation. In *VLSI Design 2000. Wireless and Digital Imaging in the Millennium. Proceedings of 13th International Conference on VLSI Design*. IEEE, 34–37.
[97] David Sheridan, Lingyi Liu, Hyungsul Kim, and Shobha Vasudevan. 2014. A coverage guided mining approach for automatic generation of succinct assertions. In *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*. IEEE, 68–73.
[98] Pouya Taatizadeh and Nicola Nicolici. 2015. Emulation-based selection and assessment of assertion checkers for post-silicon validation. In *Computer Design (ICCD), 2015 33rd IEEE International Conference on*. IEEE, 46–53.
[99] Pouya Taatizadeh and Nicola Nicolici. 2016. Automated selection of assertions for bit-flip detection during post-silicon validation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 12 (2016), 2118–2130.
[100] Pouya Taatizadeh and Nicola Nicolici. 2017. Emulation Infrastructure for the Evaluation of Hardware Assertions for Post-Silicon Validation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 6 (2017), 1866–1880.
[101] Deian Tabakov, Gila Kamhi, Moshe Y Vardi, and Eli Singerman. 2008. A temporal language for SystemC. In *Formal Methods in Computer-Aided Design, 2008. FMCAD'08*. IEEE, 1–9.
[102] Jimmy Tarrillo, Letícia Maria Bolzani Pohls, and Fabian Vargas. 2009. A hardware-scheduler for fault detection in RTOS-based embedded systems. In *2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*. IEEE, 341–347.
[103] Jason G Tong, Marc Boulé, and Zeljko Zilic. 2009. Airwolf-TG: A test generator for assertion-based dynamic verification. In *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International*. IEEE, 106–113.
[104] Jason G. Tong, Marc Boulé, and Zeljko Zilic. 2013. Test Compaction Techniques for Assertion-based Test Generation. *ACM Trans. Des. Autom. Electron. Syst.* 19, 1, Article 9 (Dec. 2013), 29 pages. https://doi.org/10.1145/2534397
[105] Babu Turumella and Mukesh Sharma. 2008. Assertion-based verification of a 32 thread SPARC™ CMT microprocessor. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*. IEEE, 256–261.
[106] Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. 2010. Goldmine: Automatic assertion generation using data mining and static analysis. In *Proceedings of the conference on Design, automation and test in Europe*. European Design and Automation Association, 626–629.
[107] DS Vidhya and Manjunath Ramachandra. 2017. A novel design in formal verification corresponding to mixed signals by differential learning. In *Computer Science On-line Conference*. Springer, 367–378.
[108] Hanbo Wang, Xingshe Zhou, Yunwei Dong, and Lei Tang. 2010. Timing properties analysis of real-time embedded systems with AADL model using model checking. In *2010 IEEE International Conference on Progress in Informatics and Computing*, Vol. 2. IEEE, 1019–1023.
[109] Hasini Witharana, Yangdi Lyu, and Prabhat Mishra. 2020. Directed Test Generation for Activation of Security Assertions in RTL Models. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* (2020).
[110] Takamitsu Yamada. 2009. Assertion generating system, program thereof, circuit verifying system, and assertion generating method. US Patent 7,603,636.