chips

MDPI

Article

A Survey on Assertion-Based Hardware Monitor Synthesis

Khitam Alatoun [1], Nikhil Saxena [2,*], Mounifah Alenazi [3] and Ranga Vemuri [4]

[1] Department of Computer Engineering, Al-Hussein Bin Talal University, Ma'an 71111, Jordan; khitam@ahu.edu.jo
[2] Department of Electrical and Computer Engineering, University of Massachusetts Amherst, Amherst, MA 01003, USA
[3] Department of Computer Science and Engineering, University of Hafr Al Batin, Hafr Al Batin 39524, Saudi Arabia; mkali@uhb.edu.sa
[4] Digital Design Environments Laboratory, University of Cincinnati, Cincinnati, OH 45221, USA; ranga.vemuri@uc.edu
* Correspondence: nikhilsaxena@umass.edu

Abstract

With the increasing complexity and connectivity of modern digital systems, verification has emerged as a critical bottleneck in the design flow. Assertion-Based Verification (ABV) has proven to be one of the most effective techniques for presilicon verification. Once assertions are generated, they can be synthesized into hardware monitors and incorporated into the design debug infrastructure. Many Design-for-Debug (DfD) methodologies leverage such hardware monitors to enhance the observability and controllability of internal system behavior, thereby accelerating verification and reducing time to market. Post-silicon debugging also benefits from the improved observability provided by these monitors. Furthermore, hardware monitors can be employed during runtime to detect and report undesired behaviors. To enable the seamless use of assertions, which are originally expressed in verification languages, throughout the entire design life cycle, several assertion synthesis approaches have been proposed. The objective of this survey is to present the existing assertion synthesis methods reported in the literature, discuss their current limitations, and identify directions for future research and improvement.

Keywords: runtime monitors; checkers; hardware monitor; synthesis assertions; assertion-based synthesis

1. Introduction

The widespread adoption of consumer electronics and IoT devices continues to drive demand for advanced semiconductor solutions, especially with the rapid growth in edge processing and AI-driven applications. This trend is accelerating semiconductor innovation and investment and increasing competitive pressure to shorten design cycles to bring products to market more quickly. With increasing the complexity of hardware designs, verification becomes critical path to production. According to the 2024 Functional Verification Study conducted by the Wilson Research Group [1], 49% of ASIC design activities are spent on verification. To address this challenge, the semiconductor industry integrates advanced and effective verification techniques into existing ASIC and FPGA design methodologies, including code coverage, functional coverage, Assertion-Based Verification (ABV), and constrained-random simulation. However, ABV is one of the most effective verification techniques and has been adopted in over 70% of ASIC designs, with a steady increase in adoption observed from 2007 to 2024 according to the Wilson Research Group study [1,2].

Academic Editors: Paris Kitsos and Gaetano Palumbo

Received: 9 March 2026
Revised: 14 May 2026
Accepted: 26 May 2026
Published: 28 May 2026

ABV is a powerful technique that allows developers to directly analyze and formally verify functional and non-functional requirements of RTL designs. It enables exhaustive coverage of large portions of the design space, without the need to develop simulation testbenches or generate test vectors [3]. In ABV, the space of possible behaviors of a design is explored mathematically using assertions. Assertions are high-level expressions often based on temporal logic that can be embedded into the design under verification (DUV) to accurately capture and check design behaviors, particularly those that span multiple clock cycles.

To enhance the observability of internal design behavior and improve the controllability of the debugging process, ABV is increasingly being extended into the Design-for-Debug (DfD) domain. This is achieved by synthesizing assertions into hardware monitors, also known as checkers, which are embedded into the design to verify security, safety, and functional correctness properties. In other words, a hardware monitor is a synthesizable Hardware Description Language (HDL) for a formal assertion. These monitors have wide applicability, including in hardware emulation, simulation acceleration, post-silicon debugging, and runtime monitoring. The hardware monitors expand the use of assertions in Integrated Circuit (IC) design from pre-fabrication to the life cycle development of ICs and throughout the lifetime of the product.

Figure 1 illustrates how hardware monitors are integrated into the hardware design flow. First, assertions are derived from hardware specifications, either manually by verification engineers or automatically using generation methods [4–6]. These assertions are typically developed in one of the two main assertion languages: SystemVerilog Assertions (SVA) or Property Specification Language (PSL). The generated assertions are then used with the Register Transfer Level (RTL) design to formally verify the design using any formal verification tools [7,8]. Subsequently, all generated assertions or a selected subset of these assertions are translated into synthesizable RTL using monitor generation tools [9–22].

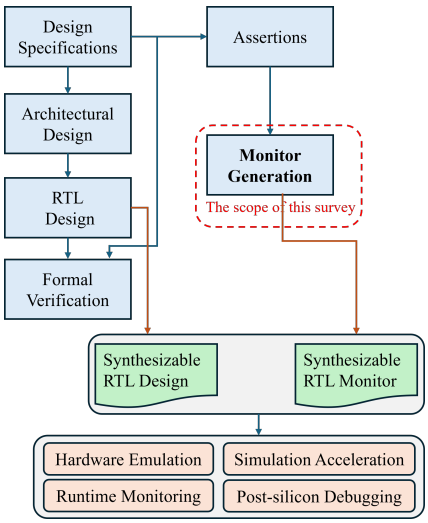

Figure 1. Hardware Design Flow with Formal Verification. Blue boxes denote pre-silicon design stages, green boxes represent synthesis step, and orange boxes highlight the post-silicon validation.

Many related works focused on using assertions in hardware verification, even on the automatic generation of assertions, or ranking assertions for hardware inclusion. The survey in [23] provides a thorough overview of the latest developments in using assertions in the hardware verification process; particularly, it focuses on how to describe system behaviors using temporal logic assertions, presents recent methods for automated assertion generation to verify both functional and non-functional requirements, and discusses techniques for validating generated assertions using test generation approaches. It also

covers utilizing assertion-based validation approaches during pre-silicon and post-silicon stages. In addition, it includes a brief section about using synthesized assertions for online monitoring and their benefits, as well as the criteria for selecting hardware monitors to be integrated into the hardware design. However, it does not discuss the techniques used for the synthesis of formal assertions into hardware monitors.

The survey presented in [24] explores the emerging field of automatic assertion mining, a technique that automatically extracts assertions from hardware designs, specifications, or execution traces. It reviews various techniques and methods to automatically mine assertions to be used in ABV for functional verification. The survey did not discuss the methods used to synthesize or translate the mined assertions into hardware monitors.

The work presented in [25] provides a review of developments and technical challenges in symbolic verification, as well as synthesis applications used in the embedded and cyber–physical systems, with a primary focus on verification approaches based on Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMTs). Their work focuses on synthesis techniques based on symbolic verification methods, while our survey specifically concentrates on the synthesis of hardware monitors from hardware assertion languages and temporal property specifications, particularly those used in digital hardware verification.

This survey paper aims to briefly introduce the assertion-based monitor generation, consolidates and classifies research works that are otherwise scattered across verification, runtime monitoring, and hardware synthesis literature. To the best of our knowledge, this is the first survey that specifically focuses on hardware monitors synthesized from formal assertions. The remainder of this paper is organized as follows. Section 2 provides an overview of assertion languages and synthesis of hardware monitor. Section 3 describes the methodology adopted for the survey. In Section 4.1, we present the main techniques used in the existing tools and list the state-of-the-art methods to assist researchers in this area. The benefit of synthesizing assertions is discussed in Section 4.2. We outline limitations in current methods in Section 4.3, the threats to validity are presented in Section 5, and finally the conclusion in Section 6.

2. Preliminaries and Overview

2.1. Assertion Language Standards

The main assertion languages used to write verification assertions are SystemVerilog Assertions (SVA) [26], Property Specification Language (PSL) [27], and Accellera Open Verification Library (OVL) [28]. SVA is the most widely used assertion language in the industry according to the Functional Verification Study conducted by the Wilson Research Group in 2024 [1]. OVL is not a separate language; it is a library of property checkers (modules) written in popular HDLs including Verilog, VHDL, and SystemVerilog. Designers can instantiate these modules to quickly verify common design properties without writing formal temporal logic from scratch. In contrast, PSL is a standalone formal language, and SVA is SystemVerilog's native assertion language. Therefore, the survey focuses on SVA and PSL languages.

From design specifications, design properties are extracted. A property is a Boolean condition that must hold true for a given design-under-test. Properties are used to create assertions, assumptions, and functional coverage statements. An assertion is an instruction to verification tools to prove whether certain design property holds true during dynamic verification (simulation) or static formal verification (model checking). During dynamic verification, if the simulator detects that an assertion is not true, it will trigger the error. During static verification, the formal property verification (FPV) tool constructs a mathematical proof wherein the design can never violate the assertion. An assumption defines expected constraints on the verification environment. In formal verification, the tool assumes that

assumption statements are true when verifying the assertions. Assumptions can be used to reduce the search space of the verification tool or to ignore impossible situations in the design. A cover statement specifies a condition in the design that must be exercised during verification.

2.1.1. System Verilog Assertions (SVA)

SVA is a standardized assertion language that is part of the SystemVerilog language, mainly used to verify the behavior of the design. In addition to checking functional correctness, SVA can be used to provide functional coverage and generate testbenches for validation. Assertions written in SVA can be embedded inline within the SystemVerilog design and testbench code, offering seamless integration into the verification environment.

SVA is structured in layers, with each layer introducing more complexity and capability. Figure 2 shows the layers with simple examples. Boolean, sequence, and property layers have no effect unless they are used within assertion statements. Booleans are standard SystemVerilog Boolean expressions used to represent logical conditions. Sequences are constructed by connecting Boolean expressions with sequence operators to describe temporal behavior. Sequence operators such as, ## which is used for delays, and [*] for repetition, specify timing and ordering in sequences. Properties combine sequences with property-level operators, such as implications ($|->$ and $|=>$), to specify behaviors that are expected to hold in a design.

Implication operators, such as $|->$ (overlapping implication) and $|=>$ (non-overlapping implication), differ in how the assertion is evaluated. Implication operators divide the assertion into two parts: the antecedent (left side) and the consequent (right side). In the $|->$ operator, if the antecedent is true in a given clock cycle, the consequent must start evaluating in the same cycle. While in the $|=>$ operator, the consequent must start evaluating in the next cycle.

Assertion statements use one of the keywords assert, assume, or cover, which cause an SVA property to be evaluated, respectively, as an assertion, assumption, or cover point.

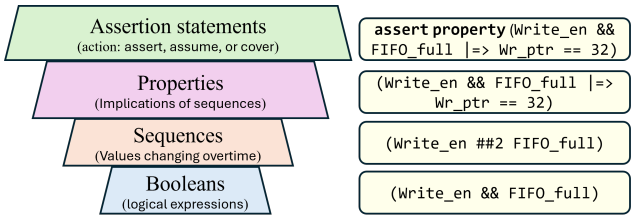

Figure 2. Layers of the SVA language.

In SVA, assertions are classified into two different types based on clocking behavior: Immediate assertion statements are simple assertion statements that are evaluated whenever they are visited in the code. They use only Boolean expressions, have no clocking or reset mechanisms, and do not support most advanced property operators.

Example:

$$Assert\ (!(FIFO\_empty\ \&\&\ FIFO\_full))$$

Concurrent assertion statements are more complex assertion statements that are used to describe behavior that spans time and are always evaluated at edges of one or more clocks. Concurrent statements are indicated by using the keyword *property* as in this example.

Example:

$$Assert\ property\ (Write\_en\ \&\&\ FIFO\_full\ |=>\ Wr\_ptr\ ==\ 32)$$

2.1.2. Property Specification Language (PSL)

PSL is a standalone specification language designed to work with multiple hardware description languages (HDLs); it has five types, including VHDL, Verilog, SystemVerilog, SystemC, and GDL. A PSL assertion expresses properties about the behavior of a design and is constructed using four levels with respect to functionality:

Boolean level, which describes events that occur over one cycle.

Example:

$$!(FIFO\_empty\ and\ FIFO\_full)$$

Temporal level, which specifies temporal relationships among Boolean expressions over time. Temporal expressions are evaluated across multiple simulation cycles.

Example:

$$always\ \{Write\_en -> next\ next\ FIFO\_full\}$$

Verification level, which has verification directives (assert, assume, . . . etc.) that tell the verification tools how to verify temporal expressions.

Example:

$$assert\ always\ \{Write\_en\ and\ FIFO\_full -> next\ \$stable(Wr\_ptr)\}$$

Modeling level, which is used to describe the behavior of design inputs as well as to represent auxiliary hardware that is not part of the design but is required for verification.

Example:

$vunit\ modeling\_level\_example\ \{$

$wire\ write\_valid;$

$assign\ write\_valid\ =\ Write\_en\ \&\&\ FIFO\_full; assert\ always\ \{write\_valid -> next\ \$stable(Wr\_ptr)\}$

$\}$

These layered constructs make PSL well-suited for writing expressive and precise assertions across various verification contexts.

2.2. Monitor Generation Example

In this section, an example of a generation hardware monitor is presented using a First In First Out (FIFO) hardware architecture. Figure 3 shows a SystemVerilog definition for a synchronous FIFO. A FIFO module includes five input signals: clock, reset, data input, write enable, and read enable. It also has three output signals: output data and two status signals $FIFO\_full$ and $FIFO\_empty$ that indicate if the FIFO is full and empty, respectively. Additionally, the FIFO uses two internal signals that serve as pointers to track the addresses of the written and read data, which are *rd_ptr* and *wr_ptr*, respectively.

One of the basic requirements in FIFO implementation is to prevent writing to the FIFO when its full. This policy can be formally verified using formal verification tools [7,8]. Figure 4 represents the corresponding SVA assertion written to capture this policy. This assertion states that whenever *Write_en* is asserted while the FIFO is full, the write pointer (*Wr_ptr*) must not change compared to its previous value. Since the assertion is not directly synthesizable, it must be translated into RTL code to enable run-time validation. Figure 5 represents the synthesized hardware circuit implementing the assertion as a monitor. In Figure 5, the *Hold* signal indicates whether the assertion is valid ($Hold = 1$) or has been violated ($Hold = 0$). In this example, the assertion is first translated into a state machine and then converted to RTL code. Figure 6 shows the generated state machine for the assertion in Figure 4. $q0$ is the initial state, while $fail$ indicates the violation of the assertion.

```
module FIFO (clk, rst, data_in, write_en, read_en,
             data_out, FIFO_full, FIFO_empty);

  input clk, rst;
  input [3:0] data_in;
  input write_en, read_en;
  output reg [3:0] data_out;
  output FIFO_full;
  output FIFO_empty;

  reg [4:0] rd_ptr, wr_ptr;
```

Figure 3. FIFO definition in Verilog.

```
assert property (@(posedge clk) disable iff(!rst) write_en && FIFO_full |=>
wr_ptr == $past(wr_ptr));
```

Figure 4. SVA assertion.

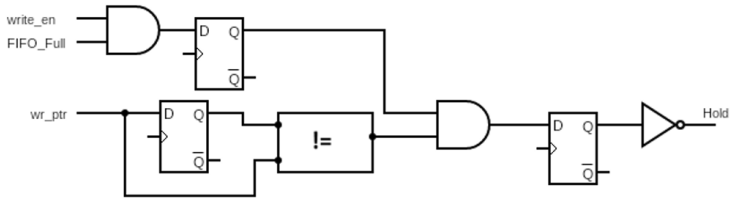

Figure 5. Hardware monitor for the SVA assertion in Figure 4.

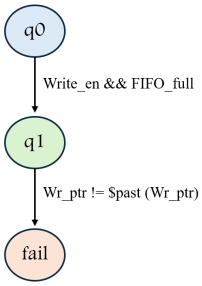

Figure 6. State machine representation for the SVA assertion in Figure 4.

3. Methodology

In this survey, we investigate the synthesis of assertions to generate hardware monitors, the benefits of using hardware monitors in the designs, and the challenges and limitations of current methods and tools.

3.1. Research Questions

The objective of this research is to understand how hardware monitors are generated from assertion statements. Thus, in this article, we define the following research questions.

RQ1: What are the fundamental techniques to synthesize assertion-based hardware monitors?

This research question aims to explain the main techniques used for synthesizing SVA or PSL assertions and generating hardware monitors.

RQ2: What is the benefit of using assertion-based hardware monitors?

This research question highlights the importance of synthesizing assertions and provides applications of hardware monitor from the literature.

RQ3: What are the challenges and limitations in the current methods for synthesizing assertion-based hardware monitors?

The research question emphasizes addressing the practical challenges of current tools in their adoption in real-world designs.

To answer these questions, we used the guidelines provided by Petersen et al. [29] to carry out a systematic mapping study. The study provides an optimal and reliable approach for documenting and analyzing current research works.

3.2. Search Criteria

For this study, we selected multiple well-established digital databases as the primary sources to ensure comprehensive coverage of relevant literature. Scopus was chosen as the primary database to construct and execute our search queries due to its comprehensive coverage of peer-reviewed literature across multiple disciplines, as well as its structured indexing and advanced query formulation capabilities. In addition to Scopus, we also included ACM Digital Library, IEEE Xplore, and Web of Science; their inclusion follows the guidelines set by Brereton et al. [30]. Using Scopus, we formulated a search string that was specific and comprehensive, ensuring the inclusion of all relevant keywords. At the same time, we avoided overly general terms in order to minimize the retrieval of irrelevant results. The final search query is:

TITLE-ABS-KEY (((“Synthesizable” OR “hardware monitor” OR “assertion checkers” OR “runtime monitor”) AND (“Assertions”))) AND (LIMIT-TO (LANGUAGE, “English”))

In addition to the automated database search, a manual search was also conducted to verify that all relevant articles had been identified and included.

Inclusion criteria: English peer-reviewed articles published at conferences, journals, and book chapters that discuss or mention models, tools, or techniques for translating PSL or SVA assertions into synthesizable HDL. Studies addressing assertion synthesis, hardware monitor generation, runtime verification synthesis, or other related formal property compilation techniques were included.

Exclusion criteria: Non-English or non-peer-reviewed articles and duplicate studies were excluded. Articles that focused solely on the utilization of hardware monitors, without addressing their generation, synthesis, translation, or implementation methodology, were also excluded. In addition, studies unrelated to assertion synthesis or formal property-to-hardware transformation were omitted.

The automated searches were conducted in September 2025. No restrictions were applied regarding publication date, and duplicate records were removed prior to the screening process. Data extraction followed an adaptive reading depth approach [29]. In this approach, the title and abstract of each article were first reviewed to determine its eligibility. When the available information was insufficient to make a decision, the full text was accessed, and additional sections were examined as needed to finalize inclusion or exclusion. After completing the screening and selection process, a total of 50 primary studies were included in the review. To ensure consistency and reliability, the quality of each study was assessed using established systematic review standards, focusing on research integrity, internal consistency, and research objectivity. Following this, data extraction and synthesis were performed on the selected studies through a structured mechanism that captured and organized key information relevant to the research objectives and questions. The extracted data included publication information, assertion specification language, synthesis methodology, monitor implementation approach, benefits of using synthesized assertions, verification strategy, and reported advantages or limitations of the synthesized methods. The collected information was then systematically organized and analyzed through descriptive synthesis, enabling a comprehensive summary of findings that forms the basis of the results presented in the subsequent section.

4. Results and Analysis

In this section, we report our results for each research question.

4.1. Existing Methods

RQ1: What are the fundamental techniques to synthesize assertion-based hardware monitors?

Synthesizing SVA or PSL assertions and generating hardware monitors has been explored by many researchers. In our review, 23 out of the 50 articles related to this research question were identified. Table 1 summarizes the results of our search process. Some rows include multiple papers because they come from the same author or research group and report related work.

Table 1. Systematic search results.

Research Database	# Results	# Unique and Related
Scopus	156	42
ACM	40	3
Web of Science	142	5
IEEE Xplore	37	0
Total (included)	**50**	

We started our search in the Scopus database and we observed that nearly all articles indexed in the ACM Digital Library, Web of Science, and IEEE Xplore were already indexed in Scopus. Research efforts in this field generally follow two main approaches: the modular method and the automata-based method. Each method provides a distinct way to translate high-level assertions into synthesizable hardware constructs. Table 2 summarizes various techniques proposed in the literature, highlighting their core methodologies and contributions. The reviewed works vary in both their optimization focus and verification targets. The optimization objectives range from the development of efficient assertion synthesis compilers and automated checker generation techniques to enhancing security coverage and reducing hardware overhead. Despite these differences, the primary goal of most approaches is to support verification and runtime monitoring in FPGA and ASIC designs. The translation of assertions into synthesizable hardware techniques are classified into three categories: modular method, automata-based method, and alternative/hybrid. In the following subsections, these methods will be discussed in detail.

Table 2. Monitor generations from the literature.

Publication	Specification Language	Type	Generated HDL	Temporal Model HDL	Optimization Focus	Verification Target
[11,31]	PSL/SVA	Automata	HDL	Sequential temporal logic with SEREs	State minimization and resource-efficient automata	Assertion checking and runtime verification
[9]	SVA	Modular	Verilog	Clocked discrete-time assertions	RTL-efficient SVA synthesis	Runtime hardware monitoring
[10]	PSL	Automata	VHDL	Sequential temporal properties	Behavioral-level PSL synthesis	Behavioral assertion verification
[22]	SVA	Modular	Verilog	Sequential temporal properties	Behavioral-level PSL synthesis	Behavioral assertion verification
[12]	PSL	Automata	VHDL	Clock-driven runtime assertions Automated	FPGA monitor synthesis	FPGA runtime verification

Table 2. *Cont.*

Publication	Specification Language	Type	Generated HDL	Temporal Model HDL	Optimization Focus	Verification Target
[13,14]	PSL	Automata	VHDL/Verilog	PSL simple subset and ranged SERE	Automata optimization	Processor security verification
[18–20,32]	PSL	Modular	VHDL	Sequential temporal properties	Automatic PSL-based prototyping	Hardware prototyping and monitoring
[17]	PSL described in Haskell	Modular	CLaSH translates Haskell to VHDL	Small subset of PSL operations	Unified language for expressing design properties and the design itself	FPGA-based hardware verification
[15]	SVA	Hybrid	Verilog	Subset of SVA assertions	Compromise between area optimization and simple implementation	FPGA-based hardware verification
[16]	SVA	Modular	Verilog	Subset of SVA assertions	Area utilization and total registers in FPGA	FPGA-based hardware verification
[21]	SVA	Automata	Verilog	Clocked SVA assertions	Offline replay and debug support	Runtime monitoring and replay analysis
[33]	SVA	Modular	Bluespec SystemVerilog (BSV)	Temporal assertions	Assertion-driven hardware synthesis	Assertion-based hardware synthesis
[34]	PSL	Alternative	VHDL	LTL-style FL operators of PSL subset	High-level synthesis of PSL assertions presented in ASM	Enhance debugging during emulation in hardware
[35]	PSL	Alternatives	NA	PSL symbolic temporal semantics	Symbolic compilation of PSL properties	Formal verification and hardware monitoring
[36]	PSL	Automata	VHDL/Verilog	Assertion-based safety temporal constraints	Logic synthesis for safety-critical optimization	Safety-critical system verification
[37]	STL	Alternative	Verilog	Subset of STL that contains past and bounded future operator	Verification mixed signal design emulation	Verification system-on-Chip (SoC) designs
[38]	SVA	Alternative	Verilog	Clocked SVA assertions	Reduce the post-silicon debug time	Cross-domain system verification
[39]	SVA	Alternative	SystemVerilog	SVA assertions with trace formats	Trace-driven checker generation (TDML/VCD)	Functional verification on hardware emulation platforms

4.1.1. Modular Method

In the modular method, each operator in the assertion is implemented in a separate synthesizable sub-component. All sub-components are connected to each other to generate a monitor circuit for the entire assertion. Das et al. developed a tool that generates a checker or monitor from an SVA assertion [9]. In this tool, sequence operators are implemented as sub-circuit that interconnect with input and output wires labeled "start" and "match", respectively. To synthesize SVA properties, the tool splits the property into its fundamental sequence expression sub-circuits and then translates them into Verilog.

The SyntHorus2 tool [19] has been developed to synthesize PSL assertions using a modular approach, where each PSL operator is implemented in a dedicated hardware component. These components are then interconnected according to the assertion's format. The tool supports only the Foundation Language (FL) subset of PSL. While its primary objective is to automatically generate RTL designs from PSL properties, SyntHorus2 can also be applied to generate monitors. It is an improved version of an earlier SyntHorus tool [32], and was later revised [18,20] to support sequential extended regular expression (SERE) operators.

Omar et al. [15] developed an SVA synthesis compiler by implementing each SVA operator in a separate Verilog module. The assertion is compiled into two levels after

parsing: the sequence level, where all required modules for sequence operators are invoked, and the property level, where all property modules are invoked. A merging unit, at the final stage, connects all produced modules based on the assertion's format.

Example.

In this section, we present an example of synthesizing a PSL assertion using the modular method. In this method, a library of primitive components, one for each PSL operator, is first developed and formulated in HDL. Figure 7 shows the architecture of a typical primitive component [40].

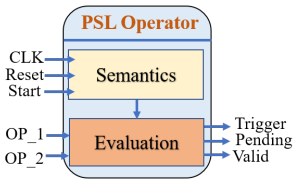

Figure 7. Architecture of primitive component.

Consider the following PSL assertion:

$P1$: **Always**$(A \mid- > \textbf{next}(B \textbf{ before } C))$;

This assertion means that whenever A signal asserts, then B must hold in the next cycle and it has to happen before C occurs. After parsing the assertion and generating its corresponding PSL expression syntax tree, the synthesis process begins by instantiating the appropriate components for each operator and connecting them according to the structure of the tree. Figure 8 shows the resulting hardware architecture generated for the assertion P1.

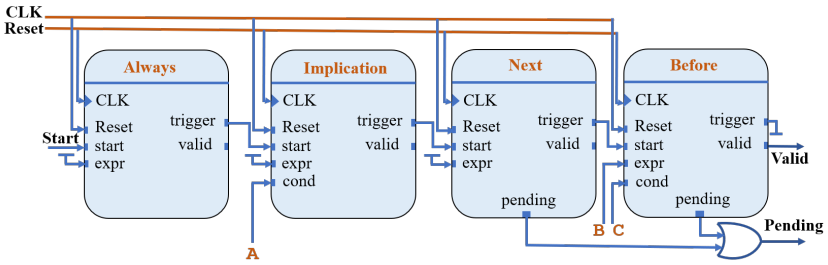

Figure 8. Monitor architecture of P1.

4.1.2. Automata-Based Method

In the Automata-based Method, the PSL or SVA assertions are represented in an intermediate automata before being converted to HDL code. The construction of automata for assertions in dynamic verification has been explored by several researchers. A. Cimatti et al. [35] converted PSL formulas into symbolically represented Nondeterministic Generalized Büchi Automata (NGBA); the work was based on the fact that PSL is built on Liner Temporal Logic (LTL) and Sequential Extended Regular Expressions (SEREs). A normal form named the Suffix Operator Normal Form (SONF) separates the LTL components and the SERE components, where each component is translated to a symbolic representation of automata. This work focused on the generation of automata from PSL properties; however, the process of converting the resulting automata into synthesizable HDL code is not addressed.

Boulé and Zillic [11] have presented the MBAC tool, an automata-based approach for synthesizing PSL sequence assertions. The tool includes a set of automata construction algorithms developed for a limited set of operators, referred to as "base cases," and then applies a set of rewrite rules to handle the remaining operators.

The SynPSL tool [10] uses a similar methodology described by Boulé and Zillic [41]. Initially, PSL formulas are reduced into base cases, called PSLmin, and then Nondeterministic Automata (NFA) is constructed for base cases. These NFAs are then converted to Deterministic Automata (DFA), which are subsequently used to generate synthesizable VHDL code. However, the SynPSL tool handles only simple Boolean expressions.

The implementation of PSL monitors is also explored by M. Jenihhin et al. [34]. The method translates PSL into Algorithmic State Machines (ASMs), then uses a high-level synthesis tool, ABELI1 [42], to convert the ASM into VHDL format. ABELI1 translates ASM into a finite state machine (FSM) as an intermediate step before generating the HDL code.

It is worth mentioning here that the use of automata to represent PSL and SVA assertions has been explored by other works, like [35], and used in many verification and analysis tools but not as hardware monitors.

Example.

In this section, we present an example of synthesizing a PSL assertion using the automata method. In this method, the automata representation of an assertion plays an intermediate role before converting it into HDL.

Consider the following PSL assertion:

$P2 : \textbf{Always}(A \mid => (B[*2] ; C));$

This assertion means: Whenever A signal asserts, it must be followed by B signal holding true for two consecutive cycles, and then C signal right after. After parsing the $P2$ assertion and generating its corresponding PSL expression syntax tree, several processes are carried out to construct the automaton representation. Figure 9 illustrates the automaton generated for the assertion $P2$. In this diagram, the red transitions indicate a violation in the assertion, and the *Fail* state is reached whenever such a violation occurs. Once the automaton is constructed, a subsequent algorithm is applied to translate it into synthesizable HDL code.

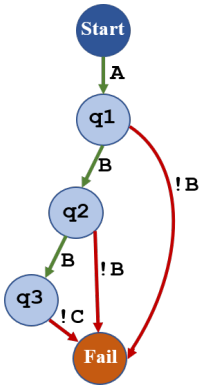

Figure 9. Automata generated for P2.

4.1.3. Hybrid/Alternative Methods

Some approaches do not conform to the automata-based or modular classification; instead, they use unique or specialized synthesis strategies that introduce alternative paradigms for monitor generation [35,37–39] or use a hybrid method [15].

For instance, the core implementation of From Signal Temporal Logic to FPGA Monitors [37] is based on the direct translation of Signal Temporal Logic (STL) operators into dedicated FPGA hardware circuits. Instead of converting specifications into finite-state automata, or converting each operator into a separate sub-component, the approach [37] maps temporal and logical operators directly into hardware components, such as comparators, counters, timers, and temporal evaluation units. The monitor operates as a streaming temporal processing architecture in which signal predicates are evaluated at every clock

cycle while timing intervals are enforced using hardware counters and temporal windows. For example, the STL property

$$G\big(A \rightarrow (B \wedge X(B) \wedge X^2(C))\big),$$

which is equivalent to the PSL in Section 4.1.2: $P2$: **Always**$(A \mid => (B[*2]\ ; C))$;, is implemented using shift registers to evaluate X and X^2, combinational logic (AND/OR/NOT), and an accumulator for G. This method focuses on continuous-time signals and real-valued system behaviors, whereas PSL and SVA are oriented toward discrete-event hardware systems and clock-driven execution models. However, it consumes significant hardware resources for complex STL formulas, such as temporal operators with large time windows or nested expressions, which may require multiple counters, buffers, and comparator chaining, increasing FPGA area and limiting scalability.

Some works employ automata within their methodology, but do not fully follow the automata-based approach [15,34,35]. For example, the work in [35] used automata, but instead of constructing explicitly enumerated finite-state automata, the approach symbolically compiles PSL specifications into symbolic transition systems and Büchi automata representations, where transitions and temporal behaviors are encoded using Boolean formulas to improve scalability and reduce state-space complexity. In [15], automata are used to translate each operator in the assertion, then the modular method is followed to combine the translated operators into a complete assertion checker.

4.2. The Importance of Assertion-Based Monitors

RQ2: What is the benefit of using assertion-based hardware monitors?

The growing complexity, size, and connectivity of modern hardware systems underscore the importance of adopting runtime monitoring to ensure the correctness, safety, and security of a design during its operational phase. Assertion-Based Verification (ABV) has recently gained traction as an effective approach to accelerate the verification process [43], especially considering that verification tasks can account for up to 51% of total development effort [43]. In ABV, verification engineers write assertions to formally capture and check design behavior. These assertions, originally developed for pre-silicon verification, are increasingly being reused in the post-silicon phase by system designers to enable ongoing validation throughout the product's life cycle.

The motivation for synthesizing these assertions into hardware monitors and carrying them into post-fab can be summarized as follows:

Pre-silicon Verification (Emulation and Formal Verification)

Hardware monitors are embedded into the design under verification (DUV) to facilitate the monitoring of property status during discrete simulations [37,44]. This technique offers key advantages over traditional formal verification methods [45], which often suffer from the state explosion problem, and oversimulation-based approaches, which can be time-consuming and require extensive testbench development. By synthesizing assertions into hardware monitors, the burden on test engineers to manually formulate correct testbenches can be significantly reduced. The work presented in [37] improves pre-silicon verification by leveraging design emulation to observe and analyze design behavior. In this approach, the hardware monitors are integrated with the design and deployed on an FPGA platform, providing a practical and efficient method for verifying correctness of the design.

Post-silicon Validation and Debugging

Traditionally, debugging hardware systems has been a complex and time-consuming task due to a substantial lack of observability and controllability of the internal operation of a design [46]. To address this challenge, hardware monitors have been incorporated into Design-for-Debug (DfD) infrastructures to enhance the observability and controllability of

the internal system during the debug process [47–54]. The work in [47] demonstrates the use of hardware assertions to detect electrically-induced errors during post-silicon validation. These errors, which often manifest as bit-flips in flip-flops, can be effectively identified using embedded hardware monitors, enabling faster and more accurate diagnosis of low-level faults. Subashree et al. [52] used hardware monitors as part of a security architecture designed for the post-silicon validation of security policies in SoCs.

Runtime (online or in-field) Monitoring

Hardware monitors are embedded into post-fab designs to observe critical properties during runtime and detect potential faults [55–59]. Such monitors are commonly referred to as online monitors, in-field monitors, or runtime monitors. For instance, the authors of [57] used a hardware monitor called the Hardware Property Checker (HPC), which is generated from PSL properties and integrated into the IC to verify critical behaviors at runtime, particularly for detecting Hardware Trojans (HTs). Many works used synthesized assertions to monitor security requirements [13,60]. In [13], hardware monitors are employed to enforce security requirements within a processor, enabling real-time detection of HTs during operation. Another application of hardware monitoring is to enhance tag security against fault attacks in Radio Frequency Identification (RFID) [55]; several hardware monitors were embedded within the RFID tag to continuously monitor its functionality and identify faults.

Additionally, to monitor more advanced design properties or security properties, such as information flow policies, the synthesized assertions can be combined with design augmentation techniques to track unexpected data propagation and detect potential side-channel leakages within the system [61–64]. This integration enables the identification of subtle security violations that may not be captured through functional verification alone.

Design augmentation techniques allow verification engineers to verify not only functional properties but also security properties like information flow properties [61,65]. For example, to verify whether signal A flows to signal Y in a multiplexer circuit (as shown in Figure 10A) (flow here means that any changes in signal A affects signal Y). This is achieved by adding tag signals and extra logic gates to the design (highlighted by the red dashed rectangle in Figure 10B). The green dashed rectangle represents the original design. If signal A differs from its tagged counterpart r_A and signal Y differs from r_Y, it indicates that a flow from A to Y exists. During the verification phase, the engineer develops this Boolean assertion ($Y == r\_Y$). To synthesize the assertion, the XOR gate in the blue dashed rectangle is added to the design. This example shows the benefit of design augmentation in verifying information properties.

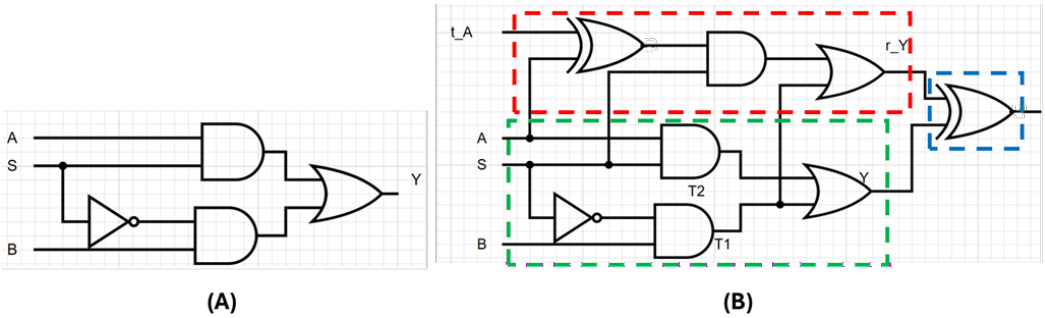

Figure 10. (**A**) 2 × 1 Mux at gate level. (**B**) Design augmentation to verify if A *flows to* Y.

4.3. Limitations in Current Methods

RQ3: What are the challenges and limitations in current methods of synthesis assertion-based hardware monitors?

Despite significant progress in assertion-based monitor synthesis, current tools still face several practical limitations that reduce their adoption in complex real-world designs.

These limitations affect the expressiveness, scalability, and applicability of the tools in various SoC architectures and design environments. This section highlights the main limitations. The limitations discussed in this section are analytical observations derived from the comparative study conducted in this survey.

Restricted Operator and Data Type: One limitation of most existing assertion synthesis tools is that they support only a subset or specific SVA or PSL operators [11,17–20,31–35]. This restriction significantly limits the expressiveness of assertions that can be synthesized into hardware monitors, particularly for complex system behaviors. To fully leverage the power of assertion-based verification and monitoring, tools should support a wider range of assertion operators, including complex temporal and sequence constructs. Additionally, comprehensive support for various signal types is essential, spanning not only Boolean and logic scalars but also vectors, integers, and enumerated types, which are commonly used in modern SoC designs.

Limited Language Feature Support in Assertion Synthesis Tools: Many modern designs like the SoC employ multiple clock domains to manage diverse functional blocks operating at different frequencies. However, most existing assertion synthesis tools are limited to handling single-clock assertions [9,11,17–20,31,32,34,35]. Supporting multiple clock domains would significantly enhance the monitoring in complex designs. The work in [40] proposes a solution for synthesizing multi-clock SVA assertions, which could potentially be extended to support PSL as well and integrated into existing tools.

Additionally, writing precise and expressive assertions often requires the use of local variables within the property itself. These local variables help simplify complex conditions, improve readability, and enable more modular assertion constructs. Therefore, assertion synthesis tools should also be capable of recognizing and correctly handling local variables in both SVA and PSL assertions to fully support the advanced features of these languages and facilitate robust assertion development.

Lack of Benchmark Assertions: The evaluation of assertion-based hardware monitor generation and its lack of standardized benchmark assertions must be addressed. Without a common benchmark suite, it becomes difficult to compare the efficiency and scalability of different tools and methodologies in a consistent and meaningful way. Researchers often use custom or ad hoc sets of assertions tailored to their specific designs [10,35,41,66], which limits the reproducibility and generalization of results. Establishing a universally adopted set of benchmark assertions that covers various property types, complexities, and domains would greatly enhance the ability to assess and improve assertion synthesis tools.

Monitor Evaluation: Evaluating the synthesized monitors requires generating an appropriate test sequence that can effectively activate and exercise the synthesized assertion logic under realistic conditions. This is crucial to ensure that the monitor behaves correctly when integrated into the system and can detect violations as intended. Model checking is one technique that can be employed for this purpose, as demonstrated in [67], where counterexamples from model checking are used to derive test inputs. However, a significant challenge arises when attempting to scale this approach to highly complex systems-on-chip (SoCs) or networks-on-chip (NoCs), where they encompass a larger scale of software and hardware components. This raises the need for more efficient, scalable test generation strategies to ensure comprehensive evaluation of runtime monitors in large-scale designs.

Monitor Optimization: Optimization of the monitors generated from the modular and automata methods is needed. Since the monitors are embedded in the design for online monitoring, it should be optimized in terms of area and power. Recent work [68] has proposed a method to optimize area and power by combining assertions, but it is limited to combining assertions with similar structure and does not provide a general framework for systematically optimizing monitors across diverse assertion types or large-scale SoC designs.

Therefore, more comprehensive and scalable optimization techniques that can reduce hardware overhead without compromising monitoring performance remains a necessity.

5. Threats to Validity

The main risks to the validity of our survey results and conclusions are researcher bias and the possibility of missing relevant articles. A threat to construct validity is our formulation of search queries and the sources of our search. To mitigate these risks, we performed searches across four key databases: Scopus, IEEE Xplore, Web of Science, and ACM to ensure we gathered the most relevant papers for our research questions. Initially, the search query was developed and tested in Scopus before being tailored to the other databases. We used keywords and their synonyms to ensure a comprehensive capture of relevant results. Furthermore, the inclusion or exclusion of papers in our survey is designed to be rigorous and objective. We mitigated possible subjective bias by following a systematic review methodology, as we have described in Section 3.

6. Conclusions

There has been growing interest in synthesizing assertions to be utilized in simulation, emulation, and online monitoring. In this survey paper, we reviewed the state of the art in assertion-based synthesis, focusing on the two primary methods used to generate hardware monitors derived from formal assertions, specifically those written in SVA and PSL. We discussed the importance of assertion monitors in modern hardware design and examined existing synthesis approaches along with their limitations. With the increasing complexity of ICs, we believe that the need to generate efficient and scalable hardware monitors will continue to grow in importance.

Author Contributions: Conceptualization, R.V. and K.A.; methodology, R.V., K.A. and M.A.; validation, N.S.; formal analysis, K.A. and M.A.; investigation, K.A., N.S. M.A. and R.V.; resources, K.A., N.S. and M.A.; data curation, K.A., N.S. and M.A.; writing—original draft preparation, K.A. and R.V.; writing—review and editing, K.A., N.S. and M.A.; visualization, K.A., N.S. and M.A.; supervision, R.V.; project administration, R.V. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Wilson Research Group IC/ASIC Functional Verification Trend Report. Available online: https://resources.sw.siemens.com/en-US/white-paper-2024-wilson-research-group-ic-asic-functional-verification-trend-report/ (accessed on 2 September 2025).
2. Wilson Research Group FPGA Functional Verification Trend Report. Available online: https://resources.sw.siemens.com/en-US/white-paper-2024-wilson-research-group-fpga-functional-verification-trend-report/ (accessed on 2 September 2025).
3. Seligman, E.; Schubert, T.; Kumar, M.V.A.K. *Formal Verification: An Essential Toolkit for Modern VLSI Design*; Elsevier: Amsterdam, The Netherlands, 2023.
4. Bjørner, N.; Browne, A.; Manna, Z. Automatic generation of invariants and intermediate assertions. *Theor. Comput. Sci.* **1997**, *173*, 49–87. [CrossRef]
5. Witharana, H.; Jayasena, A.; Whigham, A.; Mishra, P. Automated generation of security assertions for RTL models. *ACM J. Emerg. Technol. Comput. Syst.* **2023**, *19*, 1–27. [CrossRef]
6. Zhang, T.; Saab, D.; Abraham, J.A. Automatic assertion generation for simulation, formal verification and emulation. In *IEEE ISVLSI*; IEEE Computer Society: Washington, DC, USA, 2017; pp. 471–476.
7. JasperGold: Formal Property Verification Application. Available online: http://www.jasper-da.com/products (accessed on 20 September 2025).

8. Synopsys VC Formal FPV. Available online: https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html (accessed on 20 September 2025).
9. Das, S.; Mohanty, R.; Dasgupta, P.; Chakrabarti, P.P. Synthesis of SystemVerilog Assertions. In Proceedings of the Design, Automation and Test in Europe Conference, Munich, Germany, 6–10 March 2006; Volume 2, pp. 1–6.
10. Eibensteiner, F.; Findenig, R.; Pfaff, M. SynPSL: Behavioral Synthesis of PSL Assertion. In *Computer Aided System Theory—EUROCAST*; Springer: Cham, Switzerland, 2009; pp. 69–74.
11. Boulé, M.; Zilic, Z. *Generating Hardware Assertion Checkers*; Springer: Berlin/Heidelberg, Germany, 2008.
12. Sawhney, P.; Ganesh, G.; Bhattacharjee, A.K. Automatic Construction of Runtime Monitors for FPGA Based Designs. In *International Symposium on Electronic System Design*; IEEE Computer Society: Washington, DC, USA, 2011; pp. 164–169.
13. Bilzor, M.; Huffmire, T.; Irvine, C.; Levin, T. Evaluating security requirements in a general-purpose processor by combining assertion checkers with code coverage. In *IEEE International Symposium on Hardware-Oriented Security and Trust*; IEEE: Piscataway, NJ, USA, 2012; pp. 49–54.
14. Bilzor, M. Defining and Enforcing Hardware Security Requirements. Ph.D. Dissertation, Computer Science Department, U.S. Naval Postgraduate School, Monterey, CA, USA, 2011.
15. Amin, O.; Ramzy, Y.; Ibrahem, O.; Fouad, A.; Mohamed, K.; Abdelsalam, M. System Verilog Assertions Synthesis Based Compiler. In *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*; IEEE: Piscataway, NJ, USA, 2016; pp. 65–70.
16. Mohamad, N.; Ooi, C.Y.; Ismail, N.; Teh, J. SVA checker generator for FPGA-based verification platform. In *IEEE International Symposium on Circuits and Systems (ISCAS)*; IEEE: Piscataway, NJ, USA, 2016; pp. 1750–1753.
17. Uchevler, B.N.; Svarstad, K. Assertion based verification using PSL-like properties in Haskell. In *IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*; IEEE: Piscataway, NJ, USA, 2013; pp. 254–257.
18. Javaheri, F.N.; Morin-Allory, K.; Borrione, D. Synthesis of Regular Expressions Revisited: From PSL SEREs to Hardware. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2017**, *36*, 869–882. [CrossRef]
19. Morin-Allory, K.; Javaheri, F.N.; Borrione, D. SyntHorus-2: Automatic prototyping from PSL. In *Proceedings of the IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC), Istanbul, Turkey, 7–9 October 2013*; IEEE: Piscataway, NJ, USA, 2013; pp. 72–77.
20. Morin-Allory, K.; Javaheri, F.N.; Borrione, D. Efficient and Correct by Construction Assertion-Based Synthesis. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2015**, *23*, 2890–2901. [CrossRef]
21. Nho, M.; Zhou, X.; Sobelman, G.E. An SVA hardware monitor with off-line replay. In *Proceedings of the 13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT), Hangzhou, China, 25–28 October 2016*; IEEE: Piscataway, NJ, USA, 2016; pp. 1612–1614.
22. Kastelan, I.; Krajacevic, Z. Synthesizable SystemVerilog Assertions as a Methodology for SoC. In *Proceedings of the First IEEE Eastern European Conference on the Engineering of Computer Based Systems, Novi Sad, Serbia, 7–8 September 2009*; IEEE: Piscataway, NJ, USA, 2009; pp. 120–127.
23. Witharana, H.; Lyu, Y.; Charles, S.; Mishra, P. A survey on assertion-based hardware verification. *ACM Comput. Surv. (CSUR)* **2022**, *54*, 1–33. [CrossRef]
24. Iman, M.R.H.; Natale, G.D.; Morin-Allory, K. A survey on automatic assertion miners. In *2025 IEEE 28th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*; IEEE: Piscataway, NJ, USA, 2025; pp. 55–60.
25. Cordeiro, L.C.; de Lima Filho, E.B.; Bessa, I.V. Survey on automated symbolic verification and its application for synthesising cyber-physical systems. *IET Cyber-Phys. Syst. Theory Appl.* **2020**, *5*, 1–24.
26. *IEEE Std 1800–2017*; IEEE Standard for Systemverilog—Unified Hardware Design, Specification, and Verification Language, (Revision of IEEE Std 1800–2012). IEEE: Piscataway, NJ, USA, 2018; pp. 1–1315.
27. *IEEE Std 1850–2010*; IEEE Standard for Property Specification Language (PSL), (Revision of IEEE Std 1850–2005). IEEE: Piscataway, NJ, USA, 2010.
28. Accellera Standard OVL v2.8.1 Library Reference Manual. Available online: https://www.accellera.org/downloads/standards/ovl (accessed on 30 September 2025).
29. Petersen, K.; Feldt, R.; Mujtaba, S.; Mattsson, M. Systematic mapping studies in software engineering. In Proceedings of the 2th International Conference on Evaluation and Assessment in Software Engineering (EASE), BCS Learning & Development, Bari, Italy, 26–27 June 2008.
30. Brereton, P.; Kitchenham, B.A.; Budgen, D.; Turner, M.; Khalil, M. Lessons from applying the systematic literature review process within the software engineering domain. *J. Syst. Softw.* **2007**, *80*, 571–583. [CrossRef]
31. Boulé, M.; Zilic, Z. Automata-based assertion-checker synthesis of PSL properties. *ACM Trans. Des. Autom. Electron. Syst. (TODAES)* **2008**, *13*, 1–21. [CrossRef]
32. Oddos, Y.; Morin-Allory, K.; Borrione, D. From assertion-based verification to assertion-based synthesis. In *IFIP/IEEE International Conference on Very Large Scale Integration*; Springer: Berlin/Heidelberg, Germany, 2009; Volume 360, pp. 94–117.

33. Pellauer, M.; Lis, M.; Baltus, D.; Nikhil, R. Synthesis of synchronous assertions with guarded atomic actions. In Proceedings of the Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, Verona, Italy, 11–14 July 2005; pp. 15–24.
34. Jenihhin, M.; Baranov, S.; Raik, J.; Tihhomirov, V. PSL assertion checkers synthesis with ASM based HLS tool ABELITE. In *13th Latin American Test Workshop (LATW)*; IEEE: Piscataway, NJ, USA, 2012; pp. 1–6.
35. Cimatti, A.; Roveri, M.; Tonetta, S. Symbolic Compilation of PSL. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2008**, *27*, 1737–1750. [CrossRef]
36. Wenzl, M.; Fibich, C.; Rössler, P.; Taucher, H.; Matschnig, M. Logic synthesis of assertions for safety-critical applications. In *2015 IEEE International Conference on Industrial Technology (ICIT)*; IEEE: Piscataway, NJ, USA, 2015; pp. 1581–1586. [CrossRef]
37. Jakšić, S.; Bartocci, E.; Grosu, R.; Kloibhofer, R.; Nguyen, T.; Ničković, D. From signal temporal logic to FPGA monitors. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*; IEEE: Piscataway, NJ, USA, 2015; pp. 218–227.
38. Hock, O.R.; Aziz, Z.A.B.A.; Teh, J.-W. Synthesizable Assertion in Hardware and Software Development. In *Proceedings of 2017 International Conference on Imaging, Signal Processing and Communication (ICISPC), Penang, Malaysia, 26–28 July 2017*; ACM: New York, NY, USA; pp. 163–166.
39. Kayed, M.O.; Abdelsalam, M.; Guindi, R. Synthesizable SVA protocol checker generation methodology based on TDML and VCD file formats. In *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT)*; IEEE: Piscataway, NJ, USA, 2016; pp. 1–8. [CrossRef]
40. Oddos, Y.; Morin-Allory, K.; Borrione, D. Assertion-based design with Horus. In Proceedings of the 2008 6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design, Anaheim, CA, USA, 5–7 June 2008; pp. 75–76.
41. Boulé, M.; Zilic, Z. Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties. In *IEEE International High Level Design Validation and Test Workshop*; IEEE: Piscataway, NJ, USA, 2006; pp. 69–76.
42. Baranov, S. ASMs in high level synthesis of EDA tool Abelite. *FAC Proc. Vol.* **2009**, *42*, 160–165. [CrossRef]
43. Foster, H. Wilson Research Group Functional Verification Study. 2020. Available online: https://blogs.sw.siemens.com/verificationhorizons/2020/11/18/part-3-the-2020-wilson-research-group-functional-verification-study/ (accessed on 22 June 2025).
44. Pierre, L. A formal framework for testing with assertion checkers in mixed-signal simulation. In *2012 19th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2012)*; IEEE: Piscataway, NJ, USA, 2012; pp. 284–287.
45. Mettler, M.; Mueller-Gritschneder, D.; Schlichtmann, U. A distributed hardware monitoring system for runtime verification on multi-tile MPSoCs. In *ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*; IEEE: Piscataway, NJ, USA, 2020; Volume 18, pp. 1–25.
46. Sánchez, C.; Schneider, G.; Ahrendt, W.; Bartocci, E.; Bianculli, D.; Colombo, C.; Falcone, Y.; Francalanza, A.; Krstić, S.; Lourenço, J.M.; et al. A survey of challenges for runtime verification from advanced application domains. *Form. Methods Syst. Des.* **2019**, *54*, 279–335. [CrossRef]
47. Taatizadeh, P.; Nicolici, N. Automated Selection of Assertions for Bit-Flip Detection During Post-Silicon Validation. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2016**, *35*, 2118–2130. [CrossRef]
48. Boule, M.; Chenard, J.; Zilic, Z. Assertion checkers in verification, silicon debug and in-field diagnosis. In *8th International Symposium on Quality Electronic Design (ISQED'07)*; IEEE Computer Society: Washington, DC, USA, 2007; pp. 613–620.
49. Vermeulen, B.; Goossens, K. A network-on-chip monitoring infrastructure for communication-centric debug of embedded multi-processor SoCs. In *International Symposium on VLSI Design, Automation and Test*; Curran Associates, Inc.: Red Hook, NY, USA, 2009; pp. 183–186.
50. Srinivasan, S.; Jones, A.; Hingson, C.; Darrach, G.; Vemuri, R. Assertion-based trojan localization using iterative path sensitization. In Proceedings of the 2025 26th International Symposium on Quality Electronic Design (ISQED), San Francisco, CA, USA, 23–25 April 2025; pp. 1–8.
51. Srinivasan, S.; Vemuri, R. Model checking leveraged error localization for complex RTL designs. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*; IEEE: Piscataway, NJ, USA, 2022; pp. 585–592.
52. Raja, S.; Bhamidipati, P.; Liu, X.; Vemuri, R. Security Capsules: An Architecture for Post-Silicon Security Assertion Validation for Systems-on-Chip. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*; IEEE Computer Society: Washington, DC, USA, 2021; pp. 248–253.
53. Neishaburi, M.H.; Zilic, Z. An infrastructure for debug using clusters of assertion-checkers. *Microelectron. Reliab.* **2012**, *52*, 2781–2798. [CrossRef]
54. Kulkarni, A.; McElvain, K.; Larouche, M. Debugging live FPGAs using synthesizable assertions. In Proceedings of the International Engineering Consortium—DesignCon 2007, Santa Clara, CA, USA, 29 January–1 February 2007; Volume 1, pp. 246–260.
55. Mezzah, I.; Kermia, O.; Chemali, H.; Abdelmalek, O.; Beroulle, V.; Hély, D. Assertion based on-line fault detection applied on UHF RFID tag. In *8th IEEE Design and Test Symposium*; IEEE: Piscataway, NJ, USA, 2013; pp. 1–5.

56. Alsaiari, U.; Gebali, F. Hardware trojan detection using reconfigurable assertion checkers. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2019**, *27*, 1575–1586. [CrossRef]
57. Ngo, X.T.; Danger, J.; Guilley, S.; Najm, Z.; Emery, O. Hardware property checker for run-time hardware Trojan detection. In Proceedings of the 2015 European Conference on Circuit Theory and Design (ECCTD), Trondheim, Norway, 24–26 August 2015; pp. 1–4.
58. Eslami, M.; Ghasempouri, T.; Pagliarini, S. Reusing verification assertions as security checkers for hardware trojan detection. In *2022 23rd International Symposium on Quality Electronic Design (ISQED)*; IEEE: Piscataway, NJ, USA, 2022; pp. 1–6.
59. Morin-Allory, K.; Gascard, E.; Borrione, D. Synthesis of property monitors for online fault detection. *J. Circuits Syst. Comput.* **2007**, *16*, 943–960. [CrossRef]
60. Alsaiari, U.; Gebali, F.; Abd-El-Barr, M. Programmable assertion checkers for hardware Trojan detection. In Proceedings of the 2017 1st Conference on Ph.D. Research in Microelectronics and Electronics Latin America (PRIME-LA), Bariloche, Argentina, 20–23 February 2017; pp. 1–4. [CrossRef]
61. Alatoun, K.M.; Achyutha, S.M.; Vemuri, R. Efficient Methods for SoC Trust Validation Using Information Flow Verification. In *IEEE 39th International Conference on Computer Design (ICCD)*; IEEE: Piscataway, NJ, USA, 2021; pp. 608–616.
62. Alatoun, K.M.; Vemuri, R. Efficient Method for Timing-based Information Flow Verification in Hardware Designs. In Proceedings of the Great Lakes Symposium on VLSI 2022, GLSVLSI '22, Irvine, CA, USA, 6–8 June 2022; pp. 159–163.
63. Alatoun, K.; Vemuri, R. Power Side-Channel Verification in Hardware Designs. In *NAECON 2024—IEEE National Aerospace and Electronics Conference*; IEEE: Piscataway, NJ, USA, 2024; pp. 291–296.
64. Zapata, M.A.A.; Shahshahani, A.; Zilic, Z. Automated Assertion Checker Generator and Information Flow Tracking for Security Verification. In Proceedings of the 2024 25th International Symposium on Quality Electronic Design (ISQED), San Francisco, CA, USA, 3–5 April 2024; pp. 1–6.
65. Hu, W.; Ardeshiricham, A.; Kastner, R. Hardware information flow tracking. *ACM Comput. Surv. (CSUR)* **2021**, *54*, 1–39. [CrossRef]
66. Boulé, M.; Zilic, Z. Incorporating efficient assertion checkers into hardware emulation. In *2005 International Conference on Computer Design*; IEEE: Piscataway, NJ, USA, 2005; pp. 221–228.
67. Srinivasan, S.; Vemuri, R. Mutation analysis and model checking guided test generation for SoC run-time monitors. In Proceedings of the 2023 36th International Conference on VLSI Design and 2023 22nd International Conference on Embedded Systems (VLSID), Hyderabad, India, 8–12 January 2023; pp. 240–245.
68. Alatoun, K.; Shankaranarayanan, B.; Achyutha, S.M.; Vemuri, R. SoC Trust Validation Using Assertion-Based Security Monitors. In Proceedings of the 2021 22nd International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, USA, 7–9 April 2021; pp. 496–503.