Generic Configurable Checker Architecture for functional verification accelerated with AI/ML

Chandana K Nallangi (chandana.kn@intel.com),
Pavitra Balasubramanian (pavitra.balasubramanian@amd.com)
Purushothaman, Batmanabhan (batmanabhan.purushothaman@intel.com)
Intel India Private Limited, SRR2, Bangalore, India

***Abstract*- Inconsistent coding styles and checker implementations hinder verification efficiency. This leads to duplicate code, variations for similar functionalities, and difficulty in debugging. While established verification methodologies (OVM, UVM) ensure data integrity checks through scoreboards and reference models, they lack the infrastructure for standardized checker structures in control flow verification. This paper proposes a novel, standardized checkers architecture for control flow verification, promoting code readability, reducing duplication, and streamlining verification processes, that can be accelerated with AI/ML.**

I. Introduction

The verification of complex Systems-on-Chip (SoCs) with tens of thousands of features presents a significant challenge. Implementation of checkers play a crucial role in ensuring data integrity and control flow correctness for functional verification. While data integrity checks leverage standardized scoreboards[1], control flow checks[2] often lack such well-defined infrastructure.

Traditionally, control flow checks are implemented using assertions. However, these assertions can become intricate and challenging to comprehend and debug for complex features. Additionally, poorly coded assertions can impact performance and potentially lead to "vacuous passes," masking design issues. This lack of standardization motivates the need for a structured approach for control flow verification.

This paper proposes a standardized and generic architecture for verifying control flow in any design under test (DUT). Our proposed approach aims to achieve several key objectives:

- Configurability – Adaptable to accommodate design changes over time.
- Scalability – Extendible to accommodate multiple dimensions, instances of design.
- Reusability – Applicable to variations of similar designs, minimizing redundancy.
- Debuggability – Ensuring Zero Bug Escapes within defined time and available resources.
- Readability and Maintainability – Leading to robust, efficient, and maintainable test benches.
- Predictability – Promoting a consistent and reliable checker architecture that supports IP reuse.

II. Generic checker structure

Functional verification involves verifying data integrity and the sequence of operations within a design. Each operation can be decomposed into a sequence of state changes or specific signal value changes that occur at distinct points in time, as defined by the design specification. These discrete state changes can be effectively represented as an '*event*'. Consequently, verifying the relational requirements among the 'events' within each operation guarantees the correctness of the control flow.

The proposed methodology utilizes three main categories of checkers to verify these event relationships:

1) *Causal checkers:* Ensures that expected acknowledgements follow specific requests, verifying the cause-and-effect nature of the control flow.
2) *Latency checkers:* Confirms that the time intervals between events adhere to the defined timing constraints, guaranteeing proper timing behavior within the design.
3) *Event flow checkers:* Verify the correct order of events within an operation, ensuring the intended sequence of signal changes is followed.

The proposed checker structure applies a common framework for the three identified categories, promoting reusability and robust control flow verification as detailed in Figure 1. This approach introduces two key novelties:

1) *Algorithm-driven checker core generation:* The system utilizes an algorithm to automatically generate the core functionality of each checker category.
2) *Text-based checker configuration:* Users define specific checks through a text input file, simplifying configuration and potentially allowing for specification parsing with Generative AI tools. This input file can be manually created or derived from the design specifications.

The checker core leverages text files derived from Design/Architecture specifications, that capture event relationships in a predefined format. These files are parsed by the checker, enabling the dynamic creation of checkers. Events on which these checkers operate are generated and stored in a centralized event repository. Event generation utilizes information accessible from design interfaces, Registers' value changes (RAL), callbacks, and monitors. This unified approach promotes event reusability, eliminating the need for redundant event definitions.

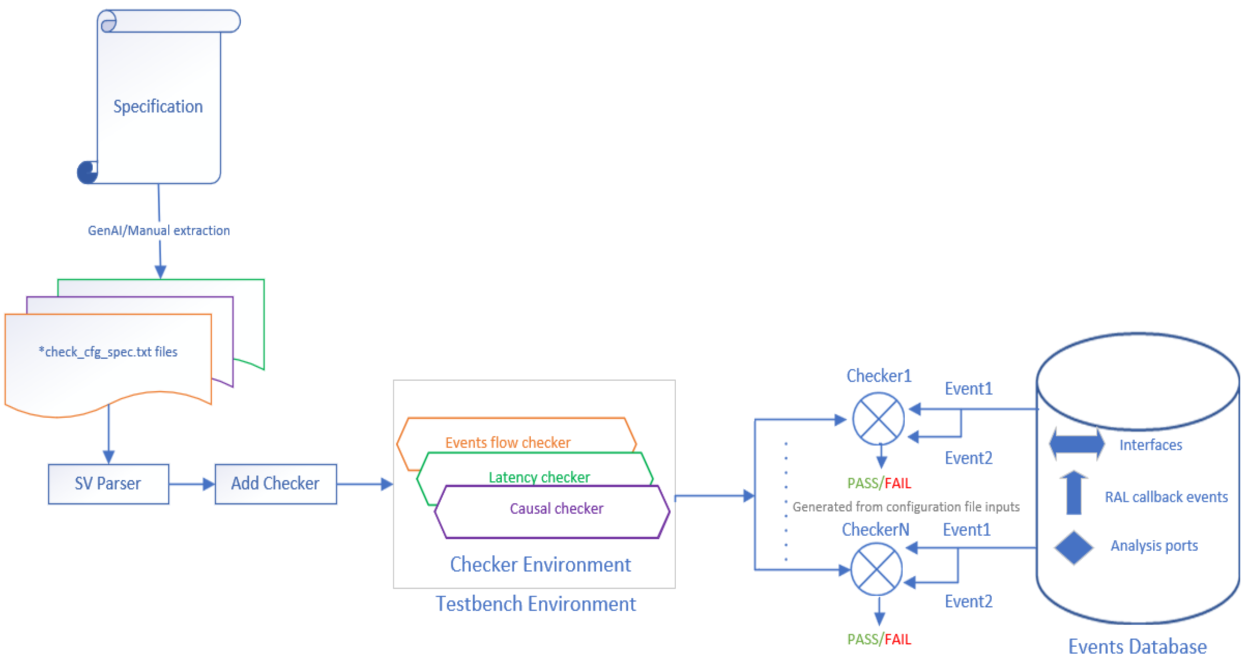

Figure 1. Generic Checker Structure for control flow operations.

A. *Causal Checker*

The Causal checker provides the infrastructure to verify the correctness of the causal operations as shown in Figure 2. Causal operations, as categorized in TABLE I, are characterized by the pairing of a request and its corresponding acknowledgement. The Request-Acknowledgment relationship supported by the checker are shown in TABLE II.

TABLE I
CAUSALITY TYPES

Causality	Description
UNIDIRECTIONAL	Sufficient. REQ => ACK, ACK can occur without REQ.
BIDIRECTIONAL	Necessary and Sufficient. REQ => ACK, ACK => REQ

TABLE II
REQUEST-ACKNOWLEDGEMENT TYPES

Type	Description
REQ_ACK	On a REQ, there should be an ACK. No new REQ before an ACK occurs.
WEAK_REQ_ACK	On a REQ, there should be an ACK. Even if REQ aborted, there should be an ACK.
REQ_ACK_ABORT	On a REQ, there should be an ACK. REQ must remain until an ACK, REQ aborted then no ACK.
MULTI_REQ_ACK	Multiple REQs can happen before an ACK. Every REQ requires an ACK.
MULTI_REQ_SINGLE_ACK	Multiple REQs can happen before an ACK. One ACK clears all REQs.

In the configuration file, users can describe the association between each request and its corresponding response. This definition includes specifying the request, its acknowledgement, the causal link between them, and the maximum duration within which the acknowledgement should be received following the request as shown in Figure 3. The checker then uses this information to verify that the system's behavior adheres to the defined temporal and causal relationships, ensuring that the system operates within the expected performance bounds.

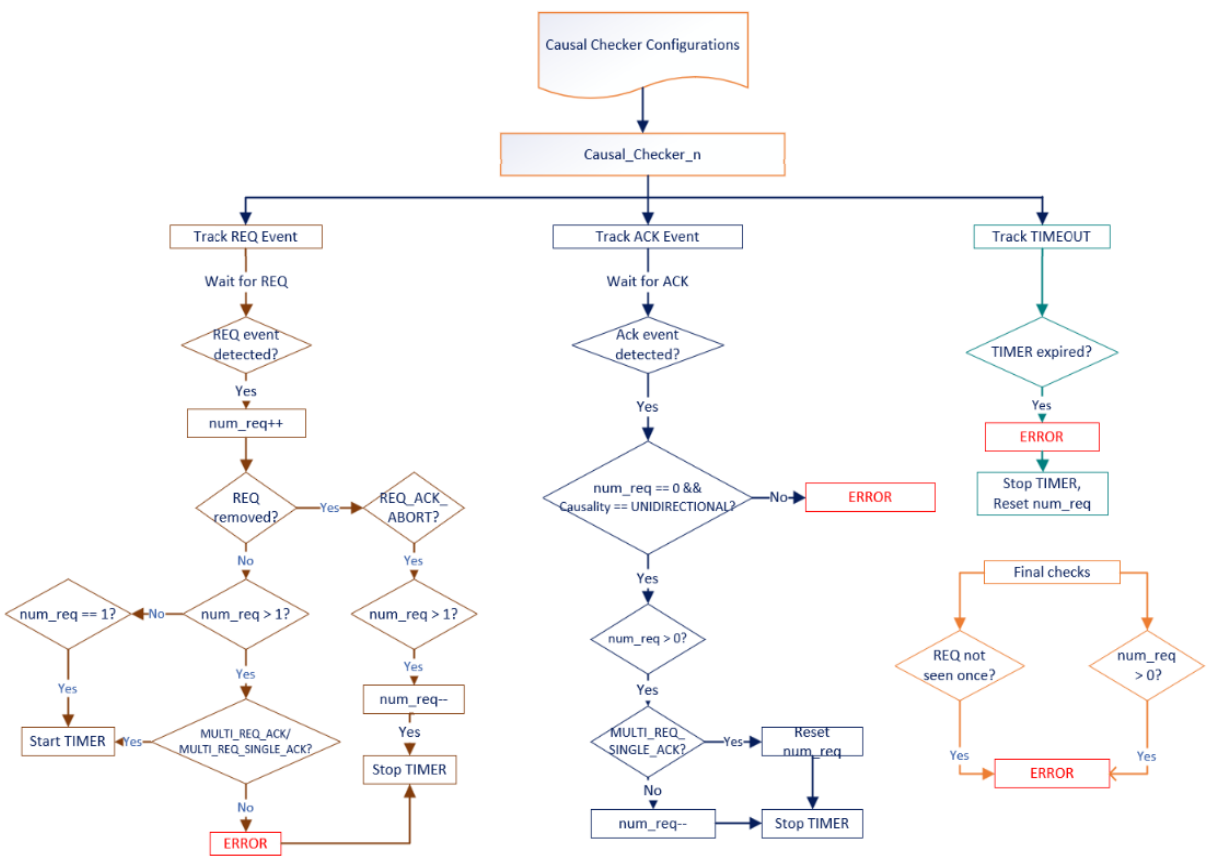

Figure 2. Causal checker flow diagram.

REQ_LABEL	ACK_LABEL	REQ_ACK_TYPE	CAUSAL_TYPE	TIMEOUT(in ns)
REQ2	ACK2	REQ_ACK	BIDIR_CAUSAL	100
REQ3	ACK3	WEAK_REQ_ACK	BIDIR_CAUSAL	150
REQ4	ACK4	REQ_ACK_ABORT	BIDIR_CAUSAL	550
REQ5	ACK5	MULTI_REQ_ACK	BIDIR_CAUSAL	350
REQ1	ACK1	MULTI_REQ_SINGLE_ACK	UNIDIR_CAUSAL	100

Figure 3. Sample input text file for Causal checker.

When the checker detects the start of an event labeled as REQ_LABEL, it begins tracking the corresponding acknowledgement event based on the configuration type. If the acknowledgment does not occur within the given time frame, the checker will raise an error as per the examples (shown in Figure 3) detailed below:

a. **REQ_ACK:** The ACK event must occur within 100ns of the start of the REQ event. Here, the checker is set to enforce bidirectional causality, and it will raise an error under two conditions: if a new REQ is identified before an ACK for the previous REQ has been observed, or if an ACK event is detected without a preceding REQ event.
b. **WEAK_REQ_ACK:** The ACK event must occur within 150ns of the start of the first REQ event. The checker will raise an error if an ACK event is not observed irrespective of the status of the REQ or if an ACK event is detected without a preceding REQ event.
c. **REQ_ACK_ABORT:** The ACK event must occur within 550ns of the start of the first REQ event. The checker will only raise an error if an ACK event is not observed when the REQ is still active.
d. **MULTI_REQ_ACK:** The ACK event must occur within 350ns of the start of the first REQ event. The checker will raise an error if an ACK event is not observed for the REQ event because of its unidirectional causality.
e. **MULTI_REQ_SINGLE_ACK:** The ACK event must occur within 100ns of the start of the first REQ event. In this case, the checker is configured with unidirectional causality, meaning it will not flag an error if an ACK event is observed without the REQ event.

B. Latency Checker

The latency checker verifies the temporal relationship between a pair of events as shown in Figure 4. It ensures that operations are executed within the designated temporal boundaries, which are specified in various metrics such as time units, clock cycles, unit intervals, or according to the rate/speed parameters of the application.

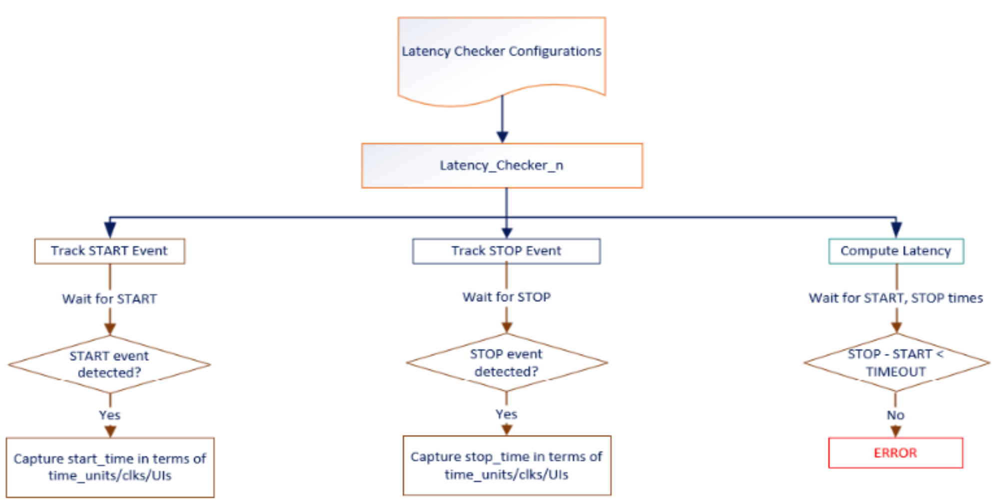

Figure 4. Latency checker Flow Diagram.

To utilize the latency checker, the user is required to specify the initiating (start) and concluding (stop) events, as well as the type of timing verification to be conducted between these events. This information should be detailed in a configuration file. The checker core is designed to dynamically interpret this data and construct a checker tailored to the provided specifications.

For instance, as outlined in the configuration file in Figure 5, when the checker identifies the commencement of an event labeled as START_LABEL, it will proceed to monitor for the corresponding stop event. If the stop event does not occur within the stipulated time frame, the checker will raise an error. The time constraints for the checks are as follows:

a. The stop event must occur within 100 microseconds (us) of the start event for the first check.
b. The stop event must occur within 10 pclks of the start event for the second check.
c. The stop event must occur within 10 Unit Intervals (UIs) of the receiver clock (rxclk) from the start event for the third check.
d. The stop event must occur within 10 UIs of the GEN2 data rate clock from the start event for the fourth check.

And the pattern continues accordingly for subsequent checks, with each check having its own defined timing requirements as per the application's needs. The checker core's dynamic parsing capability allows for flexible and automated timing verification across various scenarios as defined by the user.

```
-------------------------------------------------------------------------------------------
 START_LABEL  : STOP_LABEL   : VALUE : TIME_UNIT : CLOCK_TYPE : UIs   : RATE
-------------------------------------------------------------------------------------------
 PD_ENTER     : PD_EXIT      : 100   :    us
 PD_REQUEST   : PD_ACK       : 10    :           :   pclk
 START_EVENT1 : STOP_EVENT1  :       :           :   rxclk    :  10
 START_EVENT2 : STOP_EVENT2  :       :           :            :  10   : GEN2
 START_EVENT3 : STOP_EVENT3  :       :           :            :  100  : GEN3
```

Figure 5. Sample input text file for Latency checker.

C. Events flow Checker

Certain operations like power up, initialization etc. are deemed complete when they follow a series of events. Such features can be verified with the events flow checker as shown in Figure 6. The user must capture the order in which the events should be observed along with the relation between the events and timing information as shown in Figure 7.

The checker performs a series of rigorous checks to ensure the integrity and correctness of event sequences within a system as highlighted below:

1) *Sequential Event Validation:* The checker verifies that for each event, all preceding events that are marked as non-skippable and blocking have indeed occurred. This ensures that the event sequence adheres to the defined order without skipping necessary steps.
2) *Expected Next Event Confirmation:* For every event, the checker confirms that it is part of the expected next_event(s) list, which is specified in the configuration of the immediately preceding event. This check maintains the logical flow of events, ensuring that each event is followed by an appropriate subsequent event.
3) *Inter-Event Timing Verification:* The checker examines the timing between consecutive events. If delay check is enabled, it checks whether each event occurs within a specified timeout period from its previous event. This timeout is defined to ensure that events are not only in the correct order but also that they occur within an acceptable time frame.
4) *End-of-Test Event Coverage Check:* At the conclusion of the test, the checker reviews the sequence to confirm that all non-skippable events have taken place. This final check ensures that no critical events were missed during the test execution.

5) *Global Timeout Reporting:* The checker monitors the time elapsed since the last event and will report a timeout if there is no new event within the maximum timeout duration, which is the longest timeout defined across all event pairs. This global timeout ensures that the system does not hang indefinitely waiting for an event that may never occur.

As an example visualized in Figure 8, the checker waits for the sequence of events between EV_1.ASRT to EV_4.ASRT as per the specification requirements captured in the sample configuration file in Figure 7.

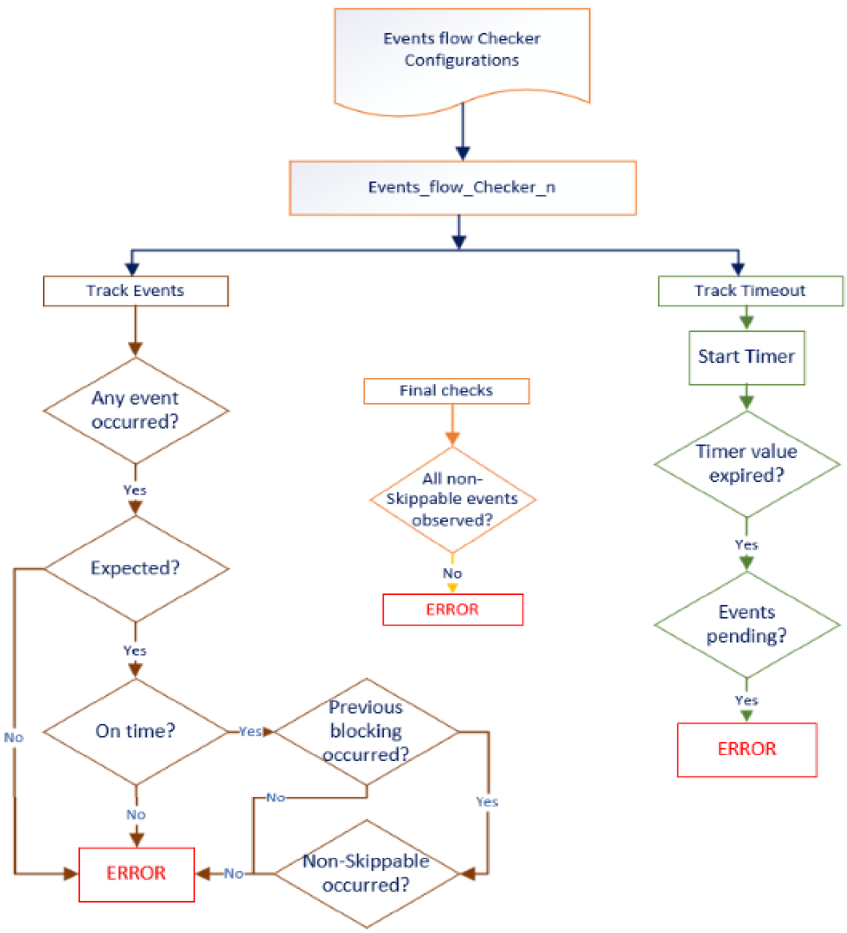

Figure 6 Events flow checker flow diagram.

id	event name	next_event(s)	IsSkippable?	IsBlocking	timeout (ns)
0	EV_1.ASRT	1,2	0	0	150
1	EV_2.ASRT	2	0	1	
2	EV_3.DEASRT	3,4			
3	EV_4.ASRT	4,5,6	0	0	
4	EV_5.ASRT	3,5,6	0	0	
5	EV_6.ASRT	3,4,6	0	1	
6	EV_4.DEASRT		0	1	

Figure 7. Sample input text file for Events Flow checker.

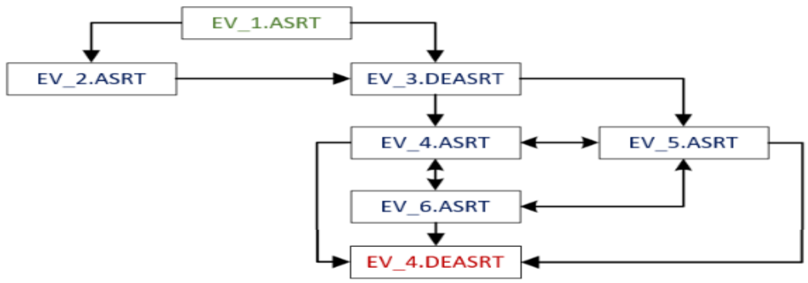

Figure 8. Example Events Flow sequence.

The checker structure also comes along with added APIs where each of the generated checkers can be enabled or disabled dynamically per instance depending on the project requirements. The configuration file follows a strict syntax to enable easier text processing. This poses limitations in automated parsing if the syntax or the formatting is not followed due to human error. The readability of configuration file can be improved by maintaining comprehensive documentation that clearly explains the intention of each of the checker entry within the file.

TABLE III
NUMBER OF CHECKERS REPLACED USING THE CONFIGURATION FILE BASED ENTRIES

Type	**Legacy checkers replaced per lane per PhyMode**
Causal Checkers	60
Latency Checkers	22
Event Flow Checkers	7

III. RESULTS AND CONCLUSION

This work introduces a novel event-based control flow verification methodology that leverages reusable checkers. This methodology has been successfully implemented in Serial I/O PHY projects for the past few years, demonstrating its practical value and effectiveness in diverse real-world applications. The text-based configuration streamlines verification setup and eliminates code redundancy in checkers. This approach has significantly reduced the code redundancy and replaced approximately 90 checkers per lane for each of the PhyModes supported by PIPE standards as shown in TABLE III. This reusability translates to substantial time and effort savings in developing and debugging verification environments for various projects and/or IP variants by maintaining the configuration files as per the requirements without the need for manually coding the checkers. They also promote consistency in verification processes, making it easier to maintain standards across different projects. Since the checker is already proven due to reusable structure, it reduces the time spent of debugging the checker itself, thereby increasing the efficiency to identify the genuine design issues.

The configurable text file can be crafted manually or generated automatically through GenAI tools, which can extract the necessary requirements directly from the specifications. This serves as a single place for adding the checkers and enables efficient reviews of the checks by the design teams improving the readability due to its English-like syntax. This ease of configuration, which is decoupled from the checker logic, allows for seamless updates to the checks without altering the checker core. Multiple IP teams can reuse the common checker core by updating the configuration files, thereby enabling the horizontal reuse of the defined methodology. The generated checkers are scalable and can be easily integrated across multiple lanes or instances within the verification environment, thanks to their plug-and-play setup.

In conclusion, this checker methodology not only fulfills its defined objectives but also sets the stage for future enhancements. The extensible core architecture is inherently equipped to support the development of new checker types, ensuring that the verification process keeps pace with the evolving demands of the design.

REFERENCES

[1] IEEE Standard for Universal Verification Methodology Language Reference Manual, IEEE Standard 1800.2-2017

[2] IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800-2023