



Schlussbericht TVB Teil 1

BMBF-Verbundprojekt VE-VIDES:

Designmethoden und HW/SW-Co-Verifikation für die eindeutige Identifizierbarkeit von Elektronikkomponenten

Zuwendungsempfänger:	Deutsches Zentrum für Luft- und Raumfahrt (DLR)
Vorhabenbezeichnung:	VE-VIDES
Förderkennzeichen:	16ME0466
Projektlaufzeit:	01.03.2021 bis 28.02.2025
Fälligkeit:	31.08.2025
Erstelldatum:	30.07.2025
Autoren:	Gregor Nitsche, Nithin Ravani Nanjundaswamy
Ansprechpartner:	Dipl.-Ing. Gregor Nitsche fon: +49 441 770507-354 E-Mail: gregor.nitsche@dlr.de

I.1 Kurzbericht

I.1.1 Motivation, Zielsetzung und Ausgangslage

Die zunehmende Komplexität eingebetteter Systeme und das Aufkommen von vertrauenswürdigen Ausführungsumgebungen (Trusted Execution Enviroments, TEEs) erfordern die Überprüfung der Integrität von Software- und Hardwarekomponenten zur Laufzeit. Insbesondere das Timing-Verhalten der Firmware, die auf eingebetteten Prozessoren wie RISC-V ausgeführt wird, ist ein kritischer Faktor, der unautorisierte Änderungen oder Anomalien in den Ausführungsmustern aufdecken kann. Kleine Abweichungen im Timing können auf Softwaremanipulationen, Leistungsengpässe oder Angriffe wie Seitenkanäle hinweisen, die möglicherweise unentdeckt bleiben. Um diese Herausforderung zu meistern, hat das VE-VIDES-Projekt ein neuartiges Konzept eingeführt, das auf statistischen Timing-Monitoren (STMos) basiert. Diese Monitore sind in der Lage, Timing-Abweichungen durch die Analyse von Laufzeit-Ereignisströmen zu identifizieren, die während der Ausführung der Firmware erfasst werden. Durch den Vergleich der beobachteten Timing-Statistiken mit vordefinierten Spezifikationen helfen STMos bei der Erkennung unbeabsichtigter Änderungen im Timing-Verhalten der Software, die die Sicherheit des Systems gefährden könnten.

Die Motivation für diese Arbeit ergibt sich aus der Notwendigkeit, die Integrität der Firmware zu überprüfen, insbesondere in Umgebungen, in denen Dritte in verschiedenen Phasen des ASIC-Designs und der Herstellung beteiligt sind. In einer solch komplexen Lieferkette reicht es oft nicht aus, sich nur auf funktionale oder formale Verifikationsmethoden zu verlassen, um zu garantieren, dass alle Laufzeitbedingungen erfüllt sind. Daher konzentriert sich dieses Projekt auf die Kombination von formalen Verifikationstechniken zur Entwurfszeit mit der Überwachung zur Laufzeit und der Integration von Observern und Monitoren in einer Weise, die die Sicherheit des Systems erhöht und gleichzeitig seine Leistung bei minimalem Hardware-Overhead aufrechterhält.

Das Hauptziel dieser Arbeit bestand darin, eine umfassende Überwachungsinfrastruktur für eingebettete Prozessoren zu entwerfen, zu implementieren und zu validieren, die speziell auf die Verifizierung des Laufzeitverhaltens unter Verwendung von Statistical Timing Monitors (STMos) ausgerichtet ist. Das Projekt umfasste mehrere Schlüsselaufgaben, darunter die Entwicklung einer domänenspezifischen Spezifikationssprache (STMoSL), das Design und die Implementierung von Observer- und Monitor-Hardwaremodulen sowie die Erstellung eines automatisierten Syntheseprozesses zur Erzeugung von STMos aus der Spezifikation. Außerdem wurde eine halbformale Verifikationsmethode angewandt, um die funktionale Korrektheit sowohl auf Simulations- als auch auf realen Hardwareplattformen sicherzustellen. Insgesamt zielten diese Aufgaben darauf ab, ein modulares und konfigurierbares Framework für die Laufzeitverifikation in eingebetteten und cyber-physischen Systemen zu schaffen.

I.1.2 Ablauf des Vorhabens, wesentliche Ergebnisse und Kooperationen

Das Projekt folgte einem strukturierten und iterativen Entwicklungsprozess, der mit der konzeptuellen Modellierung begann und sich über die Bereitstellung und Bewertung der Überwachungsinfrastruktur auf physischen Hardwareplattformen erstreckte. Zu Beginn konzentrierte sich die Arbeit auf die Entwicklung einer formalen Spezifikationssprache - STMoSL (Statistical Timing Monitor Specification Language) - zur Beschreibung des erwarteten Zeitverhaltens einer eingebetteten Firmware. Diese Sprache ermöglichte es den Benutzern, Timing-Bedingungen wie Ereignisfolgen, Jitter-Verteilungen, Beobachtungsfenster und akzeptable Abweichungsmargen auf strukturierte Weise zu definieren. Dieser Formalismus diente als Grundlage für alle nachfolgenden Synthese- und Überwachungsschritte. Nachdem die Spezifikationssprache fertiggestellt war, konzentrierte sich die nächste Phase auf die Modellierung und Entwicklung des Überwachungssystems (STMo-System). Dies beinhaltete die Entwicklung von zwei Kernmodulen: dem Observer und dem Monitor. Es wurde ein Syntheseprozess formuliert, um die RTL-Beschreibungen sowohl für die Observer- als auch für die Monitor-Komponenten automatisch auf der Grundlage der STMoSL-Spezifikation zu generieren, was die Bereitstellung vereinfacht und die Skalierbarkeit

über verschiedene Spezifikationen hinweg ermöglicht. Abbildung 1 gibt einen Überblick über die Architektur des STMo-Systems. Der linke Teil veranschaulicht den Syntheseprozess von der High-Level-Spezifikation zum RTL, während der rechte Teil die Implementierung und Integration des Monitor-Systems auf einer FPGA-Plattform zeigt.

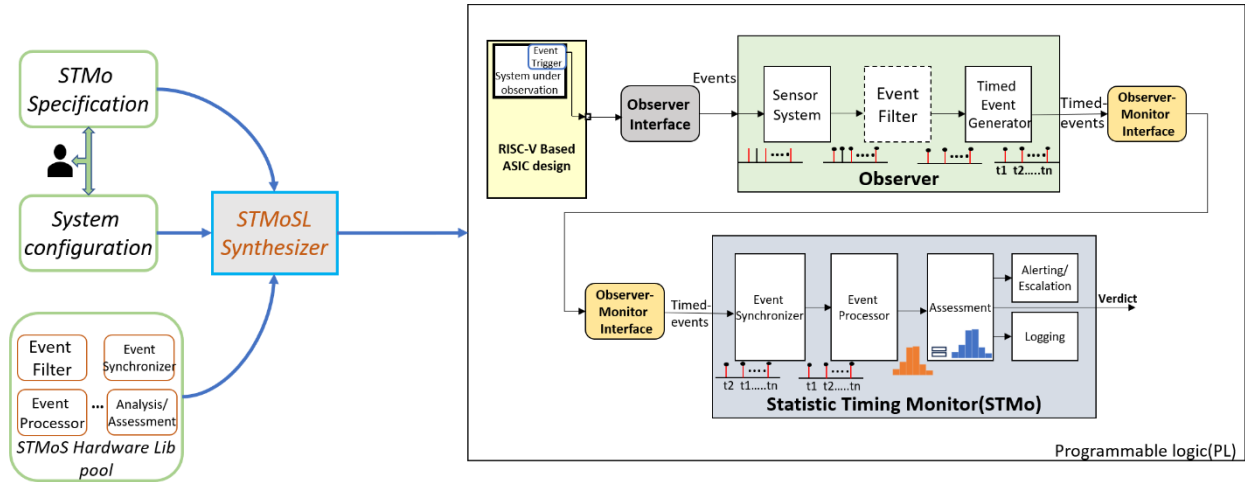


Abbildung 1: Überblick über das STMo-System (Synthese und Implementierung auf einem FPGA)

Nach der Synthese des Hardware-Designs ging das Projekt in die Verifikations- und Validierungsphase über. Dazu gehörte eine strenge simulationsbasierte Verifizierung mit verschiedenen Firmware-Szenarien, um die Korrektheit und Konsistenz der Monitorlogik sicherzustellen. Zur Erleichterung der Echtzeit-Evaluierung wurden Internal Logic Analyzer (ILA) Cores in das FPGA-Design eingebettet, so dass das Verhalten des Monitors während der Ausführung erfasst und mit der Spezifikation verglichen werden konnte.

Die letzte Phase des Projekts konzentrierte sich auf den Einsatz und die Bewertung der Überwachungsinfrastruktur auf eingebetteten Plattformen. Das komplette Überwachungssystem wurde auf einem PYNQ-Z2 FPGA-Board implementiert, auf dem ein RISC-V Softcore-Prozessor läuft. Zusätzlich wurde das System in eine vom IMMS zur Verfügung gestellte Demonstrationsplattform integriert, die die Evaluierung des STMo-Systems mit einer Echtzeit-Firmware ermöglicht. Die Monitore wurden sowohl mit synthetischen Timing-Abweichungen als auch mit realen Modifikationen, wie z.B. veränderten Ausführungszeiten in CAN-basierter Firmware, evaluiert. Die Ergebnisse zeigten, dass das System Timing-Abweichungen während der Ausführung von eingebetteter Firmware zuverlässig erkennen kann.

Zusammenfassend lässt sich sagen, dass das Projekt erfolgreich ein umfassendes statistisches Timing-Überwachungssystem entwickelt und validiert hat, das auf eingebettete Prozessoren zugeschnitten ist. Zu den wichtigsten Ergebnissen gehören die Entwicklung einer modularen Spezifikationssprache, einer automatisierten Synthese-Toolchain für die Generierung von RTL-Monitoren und einer verifizierten Beobachter-Monitor-Architektur, die in der Lage ist, Timing-Abweichungen während der Laufzeit zu erkennen und damit einen wertvollen Beitrag zum Aufbau verifizierbarer, sicherer Echtzeitsysteme zu leisten.



Schlussbericht TVB Teil 2

BMBF-Verbundprojekt VE-VIDES:

Designmethoden und HW/SW-Co-Verifikation für die eindeutige Identifizierbarkeit von Elektronikkomponenten

Zuwendungsempfänger:	Deutsches Zentrum für Luft- und Raumfahrt (DLR)
Vorhabenbezeichnung:	VE-VIDES
Förderkennzeichen:	16ME0466
Projektlaufzeit:	01.03.2021 bis 28.02.2025
Fälligkeit:	31.08.2025
Erstelldatum:	30.07.2025
Autoren:	Gregor Nitsche, Nithin Ravani Nanjundaswamy
Ansprechpartner:	Dipl.-Ing. Gregor Nitsche fon: +49 441 770507-354 E-Mail: gregor.nitsche@dlr.de

I.2.1 AP1: Anforderungen an Vertrauenswürdigkeit von IP und Designflow

Teilbeitrag: B1.3.6: Verification concept for observers and runtime monitors

Partnerbeitragsbeschreibung

This partner contribution defines the concept for the verification of observers and runtime monitors to be implemented in WP3 (B3.2.10). Based on the identified requirements, a concept is developed that is suitable for verifying the monitored properties and criteria.

Ergebnisse

This section introduces the approach for verifying the functionality of the run-time monitor through the application of semi-formal verification techniques. Formal verification is a process of proving that the designed system satisfies certain property specifications which is achieved through static analysis based on mathematical transformations of the system. This determines the correctness of the hardware or the software system. Due to the problem of state-space explosion, formal verification cannot be applied to larger designs and hence semiformal verification techniques are used. In the semi-formal verification methodology which is discussed here, both formal techniques and simulation-based model validation are tightly coupled. Abstract models with formal specifications are used for verification of the design. Then the developed design is tested on a testbench to check the correctness of the monitors behaviour. The semi-formal verification process applied in this work is conducted at two distinct phases: design-time verification and run-time verification as depicted in Figure 2.

In design-time verification, the functionality of the run-time monitor, also referred as Statistical Timing Monitors (STMo), is verified during the design-phase of the STMo components. In this approach, once the RTL components of the STMo are synthesized, behavioural simulations are performed using parameterized test benches. Further details on this approach are provided in the next section. Two other approaches, verify the STMo functionality during its run-time i.e., post implementation on the FPGA. Here the test-bench is replaced by the real firmware which provides the real-time execution-time timing behaviour to the STMo. Based on that, the first approach considers verifying the STMo results against the simulation results while the second approach considers verifying the STMo results against the results of a corresponding software-implemented STMo, being generated from the same specification. Both approaches are elaborated in the coming section.

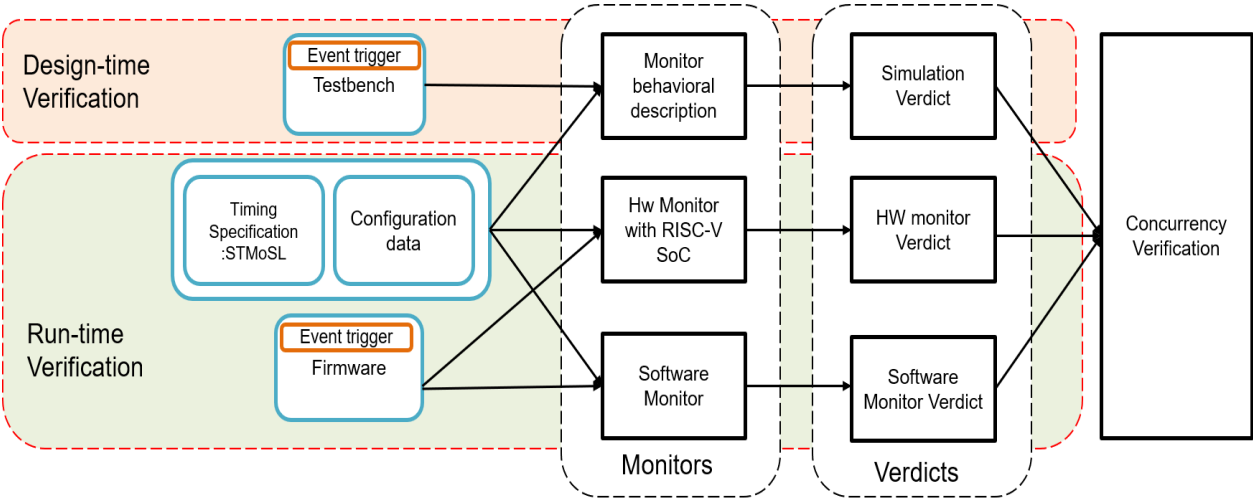


Figure 2: Concept for verification of Statistical Timing Monitors (STMo)

Design-time Verification

This approach focuses on verifying the functionality of the STMo during its development phase. Parameterized test benches are created to simulate the RTL components of the monitoring system, enabling verification of each component’s functionality as it is being designed. Events are generated within the test benches to emulate the event generation process that would rather occur in the firmware as depicted in Figure 2. An event, in this context, refers to any detectable action that arises during the execution of a target program. Further implementation details regarding the design-time verification process are provided in Section 1.2.3.

Run-time verification

The run-time verification approach assesses the functionality of the STMo during its operation following implementation on the FPGA. In this method, event generation takes place within the monitored firmware itself, rather than within a test bench. This is accomplished through event annotations embedded in the firmware, which transmit the necessary events to the observer of the STMo-system through the Observer interface. More details on the Observer and its interface are presented in Teilbeitrag: B2.3.13 Observer definition. The two distinct methods used for run-time verification are discussed in detail below.

a. Run-time Verification – in comparison with the simulation outcome

In this run-time verification approach, the functionality of the STMo system is evaluated in real time by comparing the run-time results against the simulation outcomes obtained during design-time verification. The simulation results are stored within the STMo system during the hardware generation phase and are subsequently used for real-time equivalence check during the operation. This method enables efficient verification of the STMo across a range of specifications, significantly reducing the need for manual intervention.

Figure 3 shows the representation of the equivalence check performed for the run-time outcome and the simulation outcome of the STMo-system. The events generated by the firmware running on the RISC-V core, t , should ideally be same as the events generated by the testbench, t' , as the testbench replicates the execution time behaviour of the firmware.

$$t \cong t'$$

The HW-Observer and Monitors at the pre-synthesis phase should have same functionality when compared with the HW-Observer and Monitors at run-time, provided no errors occurred during the synthesis and implementation. When event-streams produced from the testbench and the firmware are identical, the verdict from the HW-Monitors at run-time and the pre-synthesis phase will also be identical. The equivalence of the outputs between the two monitor versions verifies the correct behaviour of monitors in terms of identifying changes in timing fingerprints.

$$f(t) \cong f'(t)$$

$$g(f(t)) \cong g'(f'(t))$$

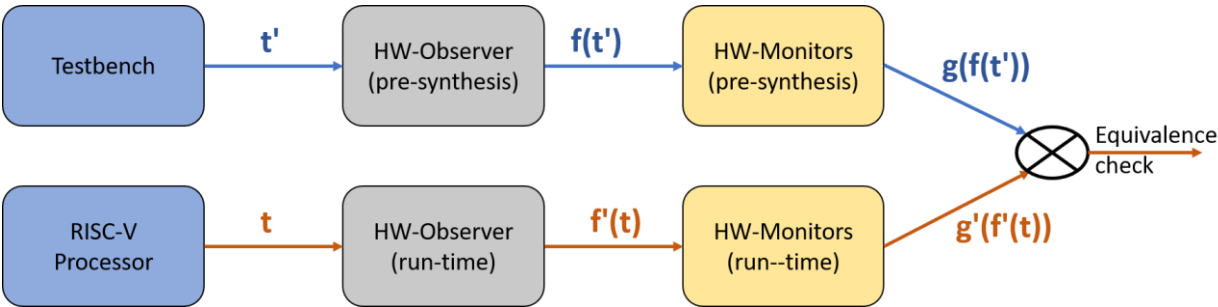


Figure 3: Equivalence check between the simulation outcome and the run-time outcome of the STMo-system

b. Run-time Verification- in comparison with software Monitor outcome

Another part of the run-time verification concept of this work follows a similar equivalence checking approach as the previous one, with the key distinction that the Statistic-Timing Monitor (STMo) results are not validated against the outcomes of a simulation but against the outcomes of a software-based STMo system corresponding to the same specification as the hardware-based implementation. That means, the software-monitor system is expected to replicate the functionality of the hardware STMo, and to produce concurrent results for a given specification for both monitoring systems.

Figure 4 shows the equivalence check consisting of a combination of observer and monitor from software and hardware implementations and comparing their results for equivalence checking. Timing events (t) of the software running on the RISC-V core serve as the input to the observers. This gives an output f(t), a tuple consisting of the block id and its respective time values. The output of the monitors is represented by g(f(t)) which is 'True' or 'False' depending on property satisfaction. Now, this model serves as the baseline reference for equivalence check in comparing the results against hardware monitors. Similar to the above diagram, the input and output of the hardware observers and monitors are represented in Figure 4. Output from the hardware observers is represented as f'(t) and the corresponding monitor outputs are represented as g'(f'(t)).

For a specific firmware that is running on the RISC-V core, generating the same event stream, the outputs of the HW and SW observers should be ideally the same (error margins taken into consideration). i.e.,

$$f(t) \cong f'(t)$$

Similarly, the verdict of both hardware and software monitors should be equivalent for a specific use case, i.e., for a given use case, the monitor output functions should be equivalent.

$$g(f(t)) \cong g'(f'(t))$$

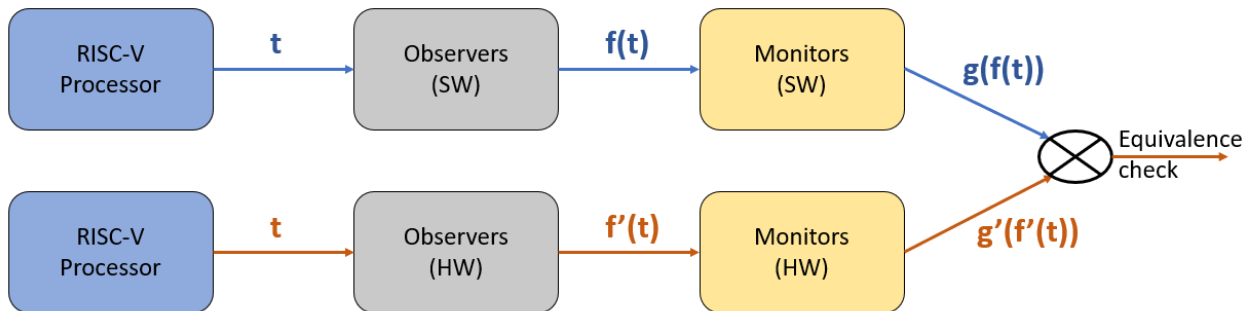


Figure 4: Equivalence checks between software and hardware STMo-systems

The equivalence of the outputs between the two monitor versions verifies the correct behavior of monitors in terms of identifying changes in timing fingerprints. Further implementation details of the concept are discussed in Section I.2.3.

I.2.2 AP2: Anforderungen an Vertrauenswürdigkeit von IP und Designflow

Teilbeitrag: B2.3.13 Observer definition

Partnerbeitragsbeschreibung

In this partner contribution, observers are developed, i.e. hardware circuits that act as probes and collect the data required by the monitors (defined in B2.3.12) of the RISC-V processor to be

monitored, making it analysable. If necessary, the observers must filter or average data in order to suppress measured values that are not required.

Ergebnisse

The section elaborates on the concept of monitor and observer, which provides a framework for assessing the timing behaviour of a software application. The fundamental principle underlying the design of monitoring framework is that any modification introduced to the software application will invariably influences its execution time, thus, making execution time a critical metric for detecting such changes. But monitoring mere execution time is not sufficient to conclude the modification to a software, since execution time also depends on various factors such as memory hierarchy, branch prediction and pipelining techniques, Input/output (I/O) Operations and so on. To tackle this, the monitor, also referred as Statistic Timing-Monitor (STMo), relies on monitoring the statistics derived from the execution time rather than the execution time itself.

The observer watches the application to obtain data that is necessary to deduce the timing behaviour and forwards this information to the monitor. Monitor, on the other hand, records the execution time statistics of the application at run-time and compares it to the predefined timing specification. By working in tandem, the observer and monitor together provides a monitoring infrastructure that helps in identification of unintended or malicious modifications to the application and ensures that the software operates within the specified timing requirements.

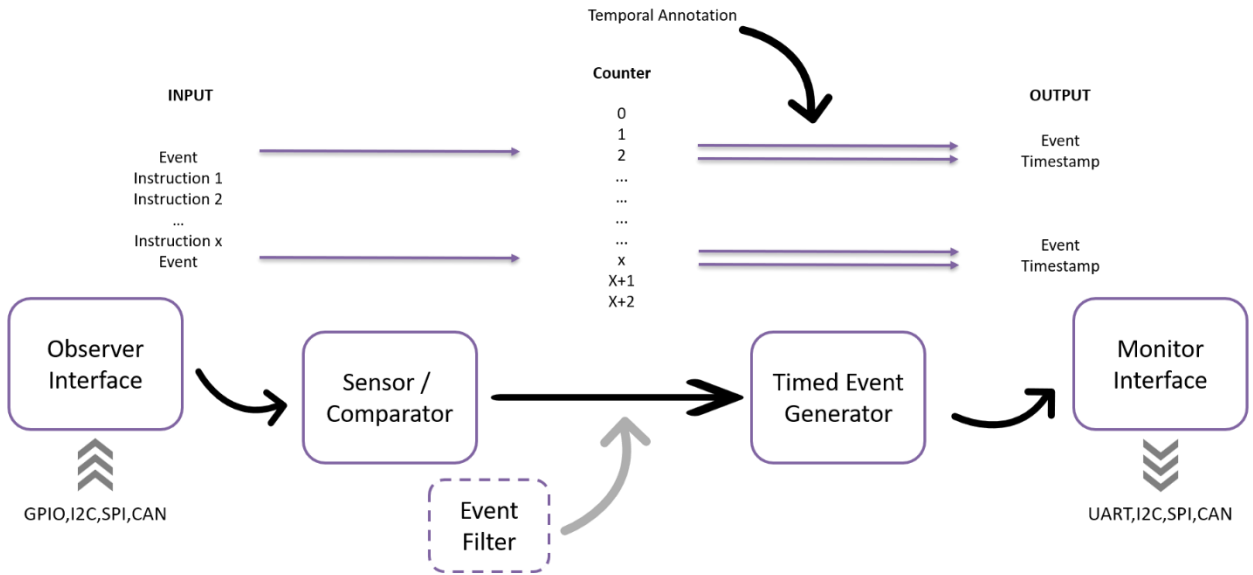


Figure 5: Components of an Observer

Figure 5 illustrates the various components integrated within an observer. The observer interface component establishes connection between the observer and the processor core, which, in this work, is a RISC-V core. The observer receives data pertaining to the timing behaviour of the software that is being executed on RISC-V. This data, which is referred as "events", represents any change in the software that is of interest to the STMo. The observer interface component can have different communication protocols, such as CAN, SPI, I2C, or GPIO, allowing developers to select the most appropriate communication medium based on their specific requirements. One important factor to consider while choosing the medium is that, the communication delay produced by the medium should be measurable and should remain constant for all events. This will ensure that all events are delayed by same amount and would not result in deducing inaccurate timing behaviour of the software. In the current work, RISC-V GPIO is used as an interface between RISC-V core and Observer since it provides a constant measurable delay and also can be operated at RISC-V core's clock frequency. This choice ensures precise and reliable timing analysis of the software under observation.

The sensor system within the observer functions as a comparator, uniquely identifying each event based on the input it receives. Once identified, these events are forwarded to the filter component, which is responsible for removing events that are not relevant for the timing monitor. The implementation of the filter can vary depending on system requirements; it can be integrated either within the observer or within the monitor.

If a large number of redundant events are being generated by the RISC-V core, placing the filter inside the observer is advantageous, as it reduces the processing burden on subsequent components by filtering out redundant events early in the process. On the other hand, if the observer needs to remain lightweight and does not have access to the specification of relevant events, it may be more appropriate to implement the filter within the monitor.

Regardless of where the filtering occurs, the events—whether filtered or unfiltered—are then passed to the timed event generator. This component utilizes a global clock to assign a timestamp to each event, marking the precise time at which the event arrived at the timed event generator. As a result, the observer outputs events accompanied by their corresponding timestamps, which are then relayed to the monitor for further processing. This system ensures that the monitor receives accurately timed events, enabling precise evaluation of the software's timing behaviour.

Teilbeitrag: B2.3.12 Monitor modelling

Partnerbeitragsbeschreibung

In this partner contribution, monitors are defined and modelled that can detect behavioural deviations in relation to a given specification. The exact properties and criteria to be monitored were defined in B1.2.11, the focus should be on functional behaviour, timing behaviour and power dissipation of RISC-V processors. This concept of monitors is to be considered as a general concept, i.e. that it should also be transferable to other processors and, in particular, it should be independent of an implementation in hardware or software at this point in time

Ergebnisse

The components that constitute the STMo are depicted in Figure 6: Components of a Statistic Timing Monitor (STMo). The timed-events are transmitted to the monitor via the Observer-Monitor interface. This interface can support various communication protocols, depending on the specific requirements of the system. For instance, if the monitor resides on the same chip as the observer, the interface may simply consist of direct wire connections. Conversely, if the monitor operates on a different chip, a full-fledged communication protocol can be necessary to facilitate the transmission.

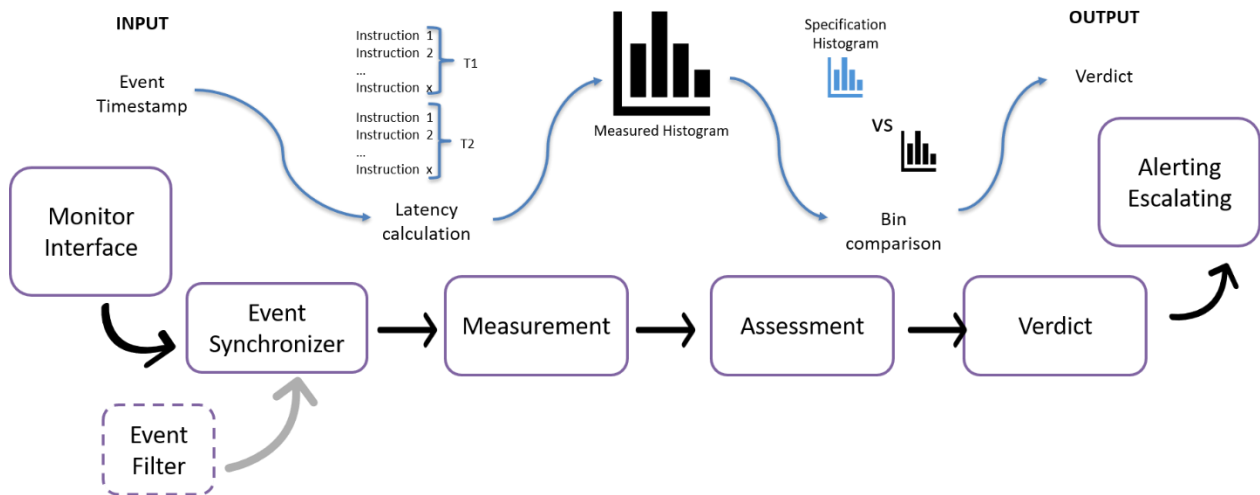


Figure 6: Components of a Statistic Timing Monitor (STMo)

Given that the events are already time-stamped, the monitor can tolerate communication delay with slight variance. In case, when the events arrive at the monitor are out of sequence, the event synchronizer block reorders the events based on their timestamps, ensuring correct temporal alignment. The ordered events are subsequently processed by the event-processor, which derives runtime timing statistics (measurements) of the software and constructs a histogram over a specified time period.

The assessment block, which plays a critical role in the monitoring process, sources a predefined histogram that represents the expected timing behaviour of the software in the absence of any malicious modifications. This reference histogram, provided by the developer, serves as the timing specification for the monitor. Further details regarding the timing specification are discussed in Section X. The assessment block then compares the real-time histogram generated by the event-processor with the specified histogram to detect any deviations from the expected timing behaviour. The outcome of this comparison serves as the monitor's verdict, indicating whether the software has been compromised. Based on this verdict, the Alerting/Escalation block will execute a developer-defined action in response to the detected anomalies. Throughout this process, logging is continuously carried-out, enabling the back-tracing of any issues that may arise within the monitoring system.

Monitor observer integration

The monitoring infrastructure, encompassing the observer and STMo, is interfaced with the RISC-V System-on-Chip (SoC) via General Purpose Input/Output (GPIO) connections as shown in the Figure 7. The system under observation, which is a software application, will be instrumented to have an event trigger. When the software execution reaches specific points of interest that are critical to the monitoring process, event trigger generates un-timed events and transmits to the Observer. In this work, event trigger is implemented as an annotation to the software to send un-timed events to the Observer through GPIO. These events are subsequently processed by Observer and STMo to obtain timing statistics through which software modification will be identified.

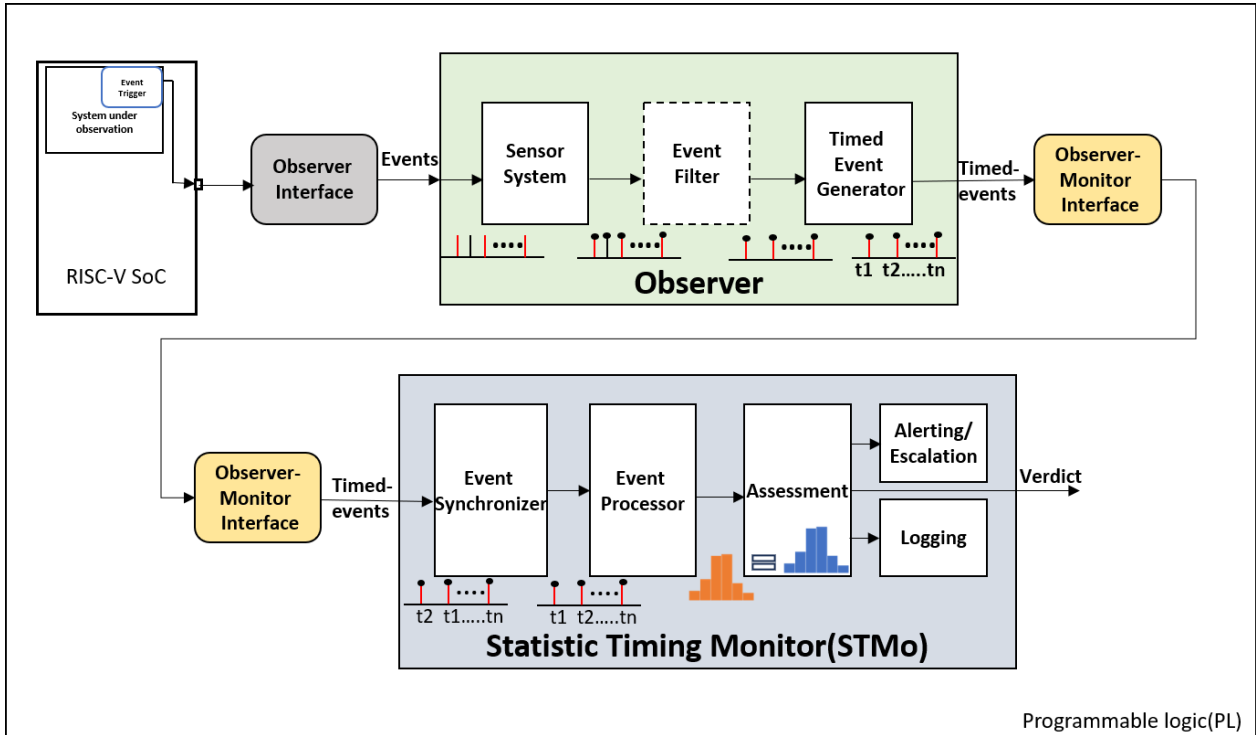


Figure 7: Monitoring system interfaced to RISC-V System-on-Chip (SoC)

Timing Specification

The assessment block inside STMo sources the expected, predefined timing behaviour histogram, which is referred to as timing specification for the Monitor. To facilitate the definition of such specification, a domain specific language (DSL) called Statistic-Time Monitor Specification Language (STMoSL) is used in this work. Using the semantics provided by the language, timing specification can be precisely defined and effectively communicated to the monitor. The semantics aids in specifying various properties of the specification histogram, each of which is individually described below.

- *Window_length* – This property defines the number of measurements required to produce a verdict. For example, if the window length is set to 100, the assessment block of the Statistic Time Monitor (STMo) will need to collect 100 measurements before making a decision.
- *Window_type* – The window type can be either sliding or looping. After the assessment block acquires the specified number of measurements (as determined by the window length), it constructs a histogram to determine the verdict. In subsequent iterations, the assessment block can either accumulate a new set of measurements equal to the window length before producing another verdict, or it can update the verdict continuously with each new measurement. The former approach is referred to as a looping window, while the latter is known as a sliding window.
- *Event_specification*- This property provides the essential information needed to uniquely identify an event. It includes the source of the event and the corresponding value associated with it, ensuring that each event can be distinguished from others.
- *Pattern_type* – The pattern type defines the relationship between events. In this work, two patterns are considered: reaction and repetitive. A reaction pattern typically involves two events and describes the occurrence of one event in relation to another. A repetitive

pattern, on the other hand, relates an event to the absolute time within the system, indicating periodic behaviour.

- *Period and offset* – This property specifies the period of occurrence of event(s) and its offset from the start of the system.
- *Jitter_distribution* – The deviation of measurements from the mean value is specified by this property. This is a critical property to construct the histogram from the measurements. For instance, if jitter distribution is specified as {40% in [-1.7ns to -1.2ns]; 40% in [-1.2ns to -0.6ns]; 10% in [-5*10⁻³s]; 10% in [5*10⁻³s]}, this means, the histogram consists of four bins, 40% of the measurements would fall within the range of -1.7ns to -1.2ns from the mean value, and similar calculations would apply for the other bins.
- *Distribution_type* – This property indicates whether the histogram is symmetric or asymmetric around its center.
- *Tolerance_level* – With this property, developer can specify the variance in the number of measurements that fall within each bin of the histogram. This property must be uniformly applied across all bins, meaning that it is not possible to set individual tolerance levels for each bin.

Using these properties, an example specification along with its mapping to the semantics is illustrated in Figure 8, and Figure 9 provides a visual representation of a histogram derived from this specification.

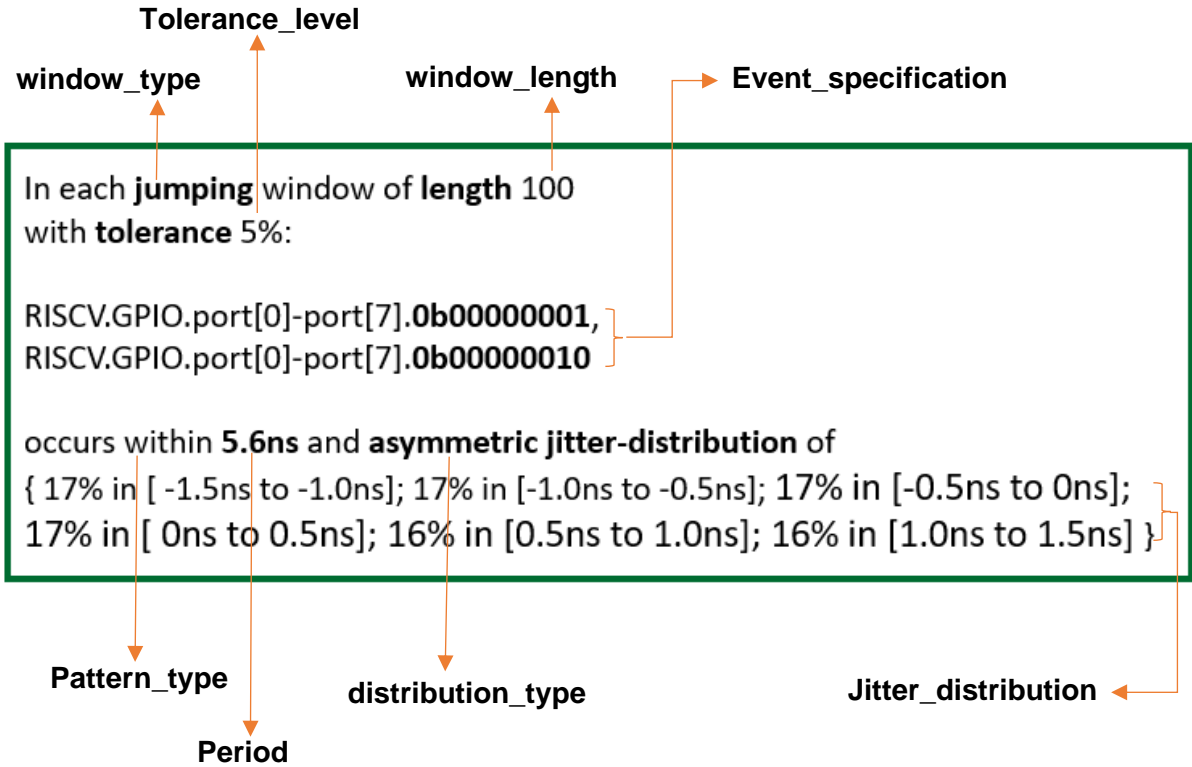


Figure 8: Example specification using STMoSL

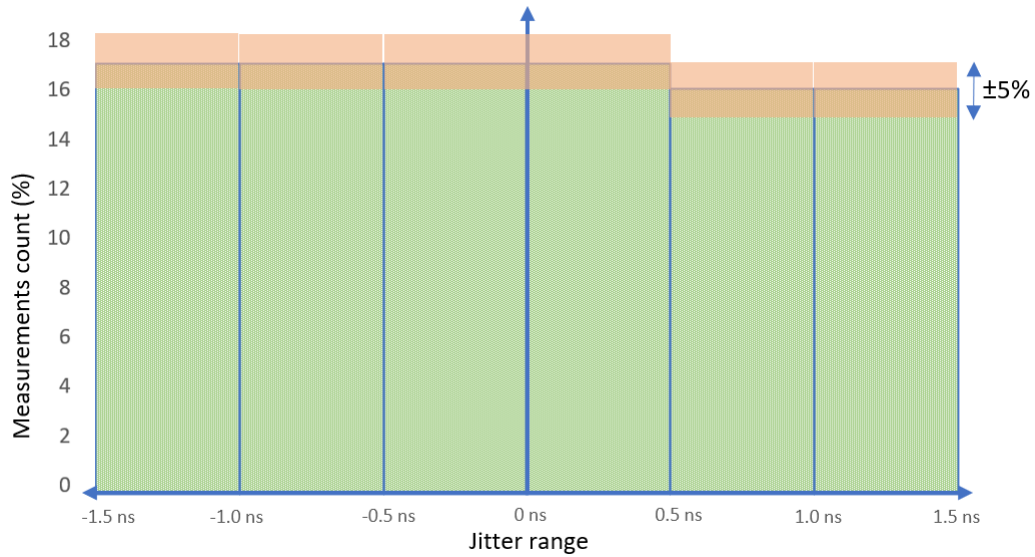


Figure 9 : Histogram derived from the example specification

Using STMoSL its possible to specify several forms of distributions. Example Specification of some of the distributions such as gaussian, exponential, binomial are shown in Figure 10, Figure 11 and Figure 12 respectively. In Figure 10 and Figure 11, a small gap between the curves and the distribution can be observed. It's possible to reduce this gap further by varying the number of distribution bins and the tolerance level and obtain a distribution as close as possible to the curve. However, it's important to consider that as the complexity of the specification increases, the amount hardware resources to implement the STMo-system also goes high.

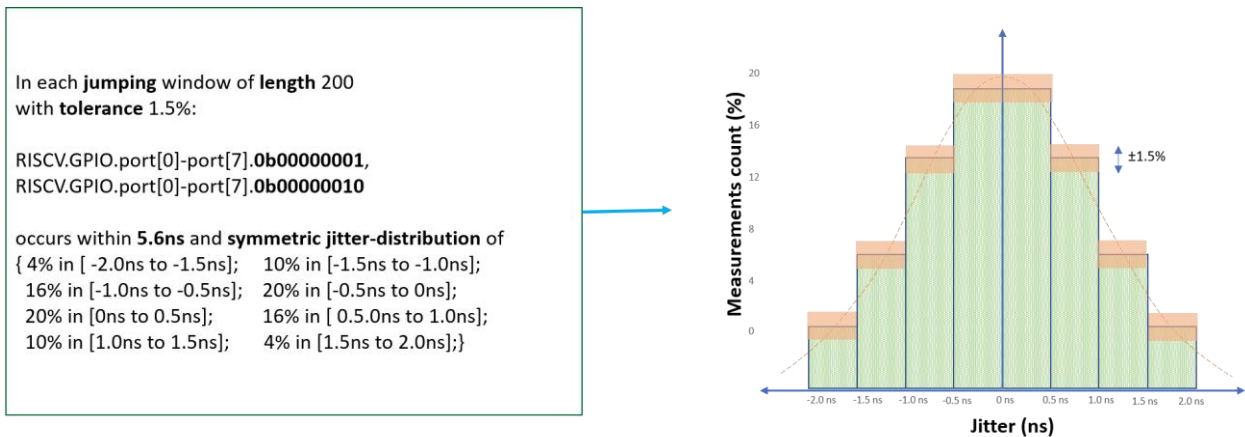


Figure 10: Gaussian distribution specified using STMoSL

In each **jumping** window of length 200 with tolerance 1.5%:

```
RISCV.GPIO.port[0]-port[7].0b00000001,
RISCV.GPIO.port[0]-port[7].0b00000010
```

occurs within 5.6ns and symmetric jitter-distribution of

{ 25% in [-2.0ns to -1.5ns];	20% in [-1.5ns to -1.0ns];
16% in [-1.0ns to -0.5ns];	13% in [-0.5ns to 0ns];
10% in [0ns to 0.5ns];	7% in [0.5.0ns to 1.0ns];
5% in [1.0ns to 1.5ns];	4% in [1.5ns to 2.0ns];

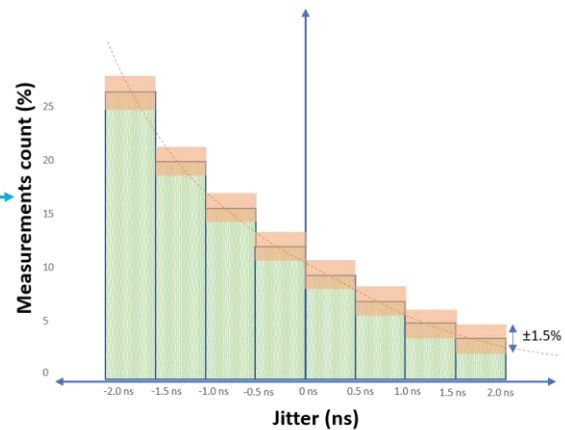


Figure 11: Logrithm distribution specified in STMoSL

In each **jumping** window of length 200 with tolerance 1.5%:

```
RISCV.GPIO.port[0]-port[7].0b00000001,
RISCV.GPIO.port[0]-port[7].0b00000010
```

occurs within 5.6ns and symmetric jitter-distribution of

{ 12.5% in [-2.0ns to -1.5ns];	12.5% in [-1.5ns to -1.0ns];
12.5% in [-1.0ns to -0.5ns];	12.5% in [-0.5ns to 0ns];
12.5% in [0ns to 0.5ns];	12.5% in [0.5.0ns to 1.0ns];
12.5% in [1.0ns to 1.5ns];	12.5% in [1.5ns to 2.0ns];

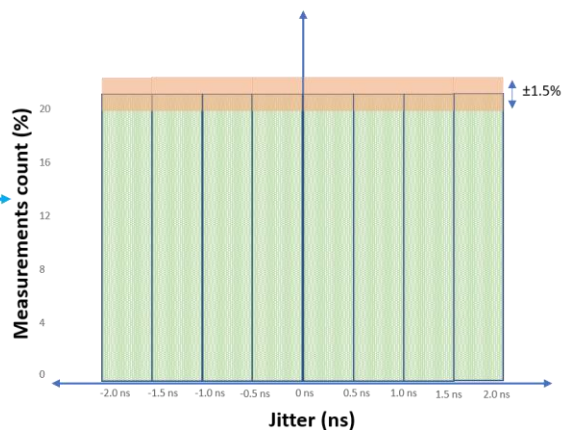


Figure 12: Uniform distribution specified in STMoSL

Teilbeitrag: B2.3.14 Hardware components for monitor implementation

Partnerbeitragsbeschreibung

In this partner contribution, the monitors that are defined and specified in B2.3.12 are designed as hardware circuits together with the connection to the observers that are defined in B2.3.13. These hardware circuits can be configurable, parameterizable or even fully programmable and are implemented as simulation components and prototype RTL components.

Ergebnisse

The various sub-components of the Observer and the Monitor, which constitutes the STMo-System is implemented in VHDL as hardware modules. The implementation details of the sub-modules along with their flowchart is provided below.

Sensor/Comparator

The sensor/comparator sub-system within the observer functions as a comparator by uniquely identifying each event from the input received. The observer interface transmits an event stream (event_stream) containing event-related information generated by the firmware. The sensor/comparator sub-module extracts relevant event details from this stream. In this implementation, the observer interface utilizes a 7-bit GPIO module, resulting in a 7-bit wide data stream received by the sensor/comparator.

On detecting any change in the data stream, the comparator captures the updated value and assigns it to the event signal, which serves as its output. Additionally, each detected change in the data stream is flagged via the event_marker signal, which is also propagated to subsequent sub-components. Figure 13 presents the flowchart illustrating the implementation of the sensor/comparator sub-module.

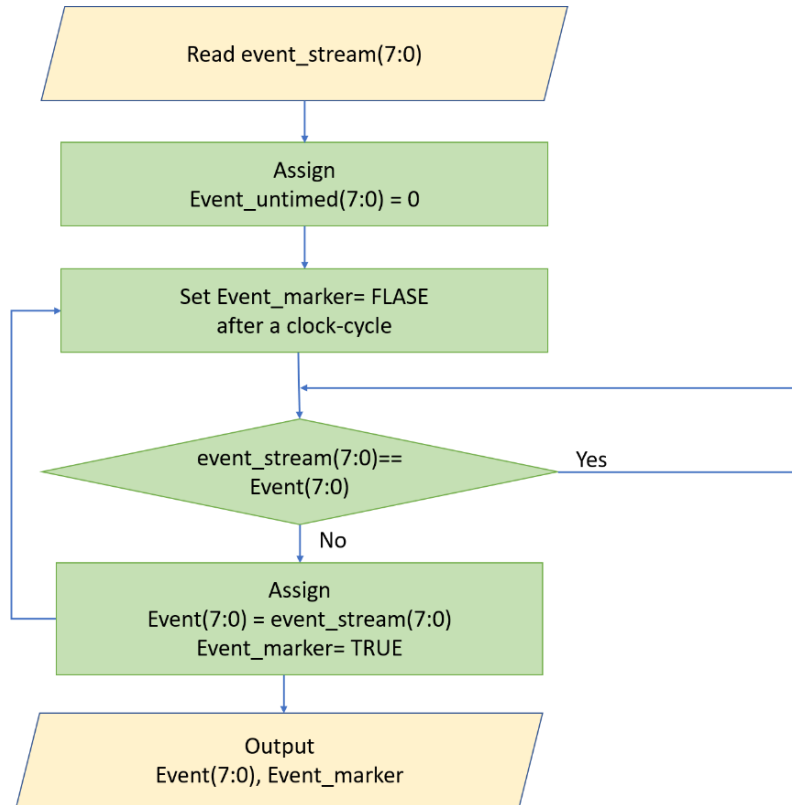


Figure 13: Flowchart for the implementation of Sensor/Comparator

Event Filter

The output generated by the sensor/comparator module may include events that are not relevant for system monitoring. To ensure only pertinent events are processed, an event filter is employed. This filter is responsible for eliminating extraneous events and retaining only those defined in the STMo-Specification.

The placement of the event filter—either within the Observer or the Monitor—is determined based on configuration parameters provided by the user. Factors influencing this decision include the availability of hardware resources allocated for the Observer and the volume of events generated by the firmware.

Functionally, the event filter receives input from the sensor/comparator module and reads the STMo-Specification. Upon detection of an event (signalled by the even_marker being set to true by the comparator), the filter verifies whether the incoming event matches any event in the STMo-Specification. If a match is found, the event is passed to a new signal, Event_filtered; otherwise, the event is discarded. This mechanism ensures that only specification-compliant events progress to subsequent modules, thereby improving the efficiency of the monitoring process. The operational flow of the event filter module is illustrated in Figure 2.

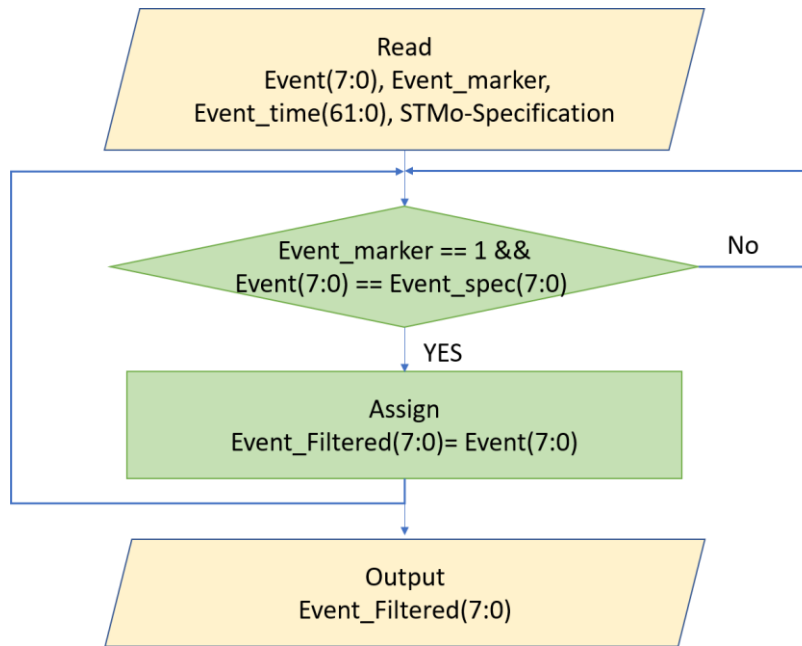


Figure 14: Flowchart for the implementation of Event Filter

Timed-Event Generator

The final module within the Observer is the Timed-Event Generator, which is responsible for associating temporal information with each event. Up to this stage, the system handles events in a time-agnostic manner. The role of the Timed-Event Generator is to assign precise time-stamps to events, thereby enabling analysis of the firmware's timing behavior.

Accurate time-stamping is critical to understanding the temporal characteristics of firmware execution. To this end, it is essential that each event is assigned a time-stamp that is as close as possible to its actual time of occurrence. Moreover, maintaining a constant and deterministic delay between event generation and time-stamping is crucial. This consistency ensures that when differential timing analysis is performed, the effect of the delay becomes nullified.

As illustrated in Figure 15, the Timed-Event Generator receives either filtered or unfiltered events, along with the current time from a global clock source. Upon detecting an incoming event, the module immediately assigns the current global time to a 64-bit signal named `Event_time`. This process is repeated for every event, ensuring that each output consists of the event itself along with its corresponding time-stamp.

By systematically assigning time information to all events, the Timed-Event Generator enables precise temporal tracing and monitoring of firmware execution

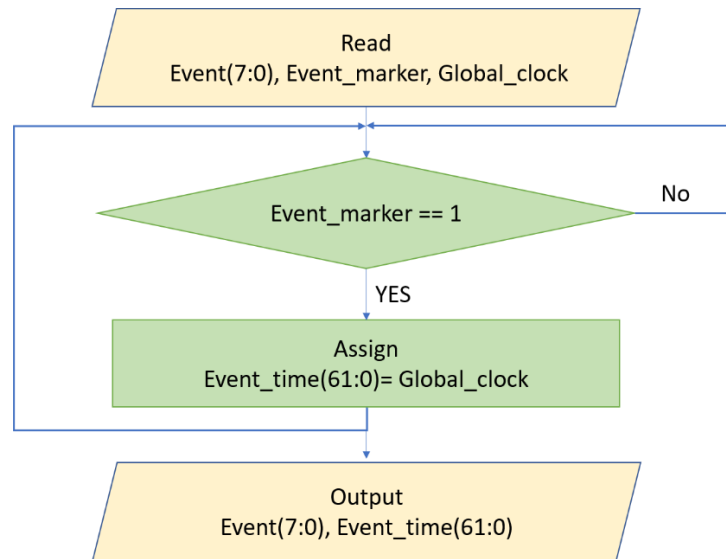


Figure 15: Flowchart for the implementation of Timed-Event Generator

Synchroniser

The Synchronizer is the first module that resides within the Monitor. Its primary function is to ensure temporal ordering of events before any timing analysis is performed. In scenarios where communication between the Observer and the Monitor introduces unpredictable delays, events may arrive out of order at the Monitor. This disorder will lead to inaccurate temporal measurements. To address this, the Synchronizer reorders the incoming timed events based on their time-stamps, restoring the correct temporal sequence.

As illustrated in Figure 15, the Synchronizer is implemented using the Quick Sort algorithm, which sorts events in ascending order based on their time-stamps generated by the Timed-Event Generator module. Upon receiving an event—indicated by the `even_marker` signal from the comparator module—the Synchronizer classifies the event time-stamps into one of two time-stamp arrays based on its type:

- **Start Events:** Includes events marking the start of a reaction pattern. For a repetitive pattern, all events time is assigned to start event time-stamp array as temporal measurements are carried out for occurrence of a single event.
- **Stop Events:** Includes events denoting the end of a specified reaction pattern.

Once the number of collected events matches the predefined window size, the Quick Sort function is applied to each array to sort the events in based on its time-stamp. A signal, `Is_sorted`, is then asserted to inform downstream modules that sorting is complete and the data is ready for further processing.

While the Synchronizer introduces an additional processing delay, it becomes essential when communication latency is non-deterministic. This ensures reliable and meaningful timing analysis despite transmission delays. The delay introduced by the Synchronizer is primarily influenced by two factors:

1. **Window Size:** Larger window sizes require more events to be accumulated before sorting can begin, resulting in increased latency.
2. **Communication Delay:** Any latency in transmitting events from the Observer to the Monitor directly contributes to the total delay.

The maximum possible delay introduced by the Synchronizer is approximately equal to the time taken to receive all required events within a window, plus the worst-case communication delay.

For detailed information regarding window size and specification patterns, please refer to the Timing Specification section.

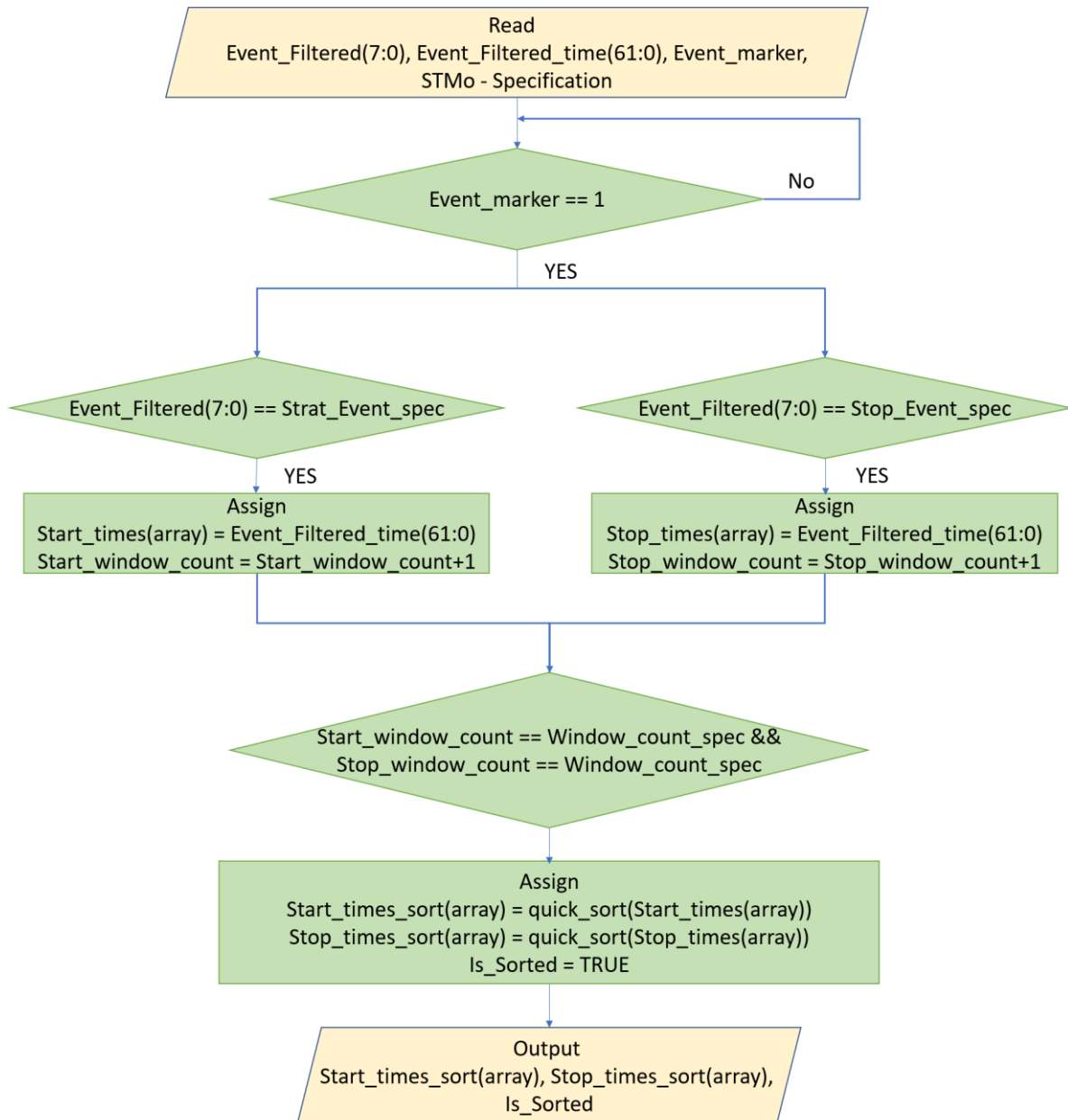


Figure 16: Flowchart for the implementation of Synchronizer

Measurement

The Measurement Module is a critical component within the Monitor, responsible for performing temporal calculations based on the sorted event data. Specifically, it computes latency between events for reaction and repetitive pattern and assigns the computed latencies to the respective bins as defined in the STMO-specification. These bin assignments are essential for the subsequent evaluation carried out by the Assessment Module.

The operation of the Measurement Module is illustrated in Figure 17 which reads the sorted event arrays provided by the Synchronizer, as well as the pattern type defined in the STMO-Specification. When the Is_sorted marker signal is set to true by the synchronizer, the module checks if the pattern type is reaction or repetitive.

- For reaction patterns, it calculates the latency as the difference between corresponding time-stamps in the sorted start and stop event arrays.
- For repetitive patterns, only the sorted start array is considered, and latency is computed as the difference between successive time-stamps.

After all relevant latencies are computed, the next step involves binning each latency value. This is achieved by comparing each latency against the bin thresholds defined in the specification and incrementing the corresponding bin count in the bin_measurements array. Once all latency values are assigned to their respective bins, the module sets the Asses signal to true, signalling the Assessment Module that it can proceed with its evaluation process.

Assessment

The Assessment Module plays a vital role in verifying the timing behaviour of the firmware against the specification provided by the user. Its primary function is to evaluate whether the observed temporal characteristics of the system comply with the user defined STMo-Specification.

As illustrated in Figure 18, the module reads the bin measurement data from the Measurement Module. The assessment process is triggered once all latency values for a given window have been categorized into their respective bins, which is indicated by the assertion of the Asses signal.

Upon activation, the Assessment Module compares each measured bin count against the corresponding reference bin count defined in the specification. For each bin, the absolute difference between the measured and specified counts is computed. If this difference is less than or equal to the defined tolerance threshold, the timing behaviour within the current window is considered compliant. This result is reflected in the assessment_outcome signal:

- A true value indicates that the timing behaviour is within acceptable bounds.
- A false value signals a deviation from the expected temporal behaviour.

Depending on the user-defined requirements, the assessment_outcome can either be used directly as a verdict or serve as an intermediate result. If a single-window deviation is sufficient to determine a violation, the outcome can be treated as a final decision. Otherwise, a separate Verdict Module can be introduced to aggregate results over multiple windows. In this work, a Verdict Module was implemented to consider the last three outcomes from the Assessment Module. A positive final verdict is produced if at least two out of three outcomes indicate compliance, thereby providing a majority-based decision mechanism for increased robustness against transient anomalies.

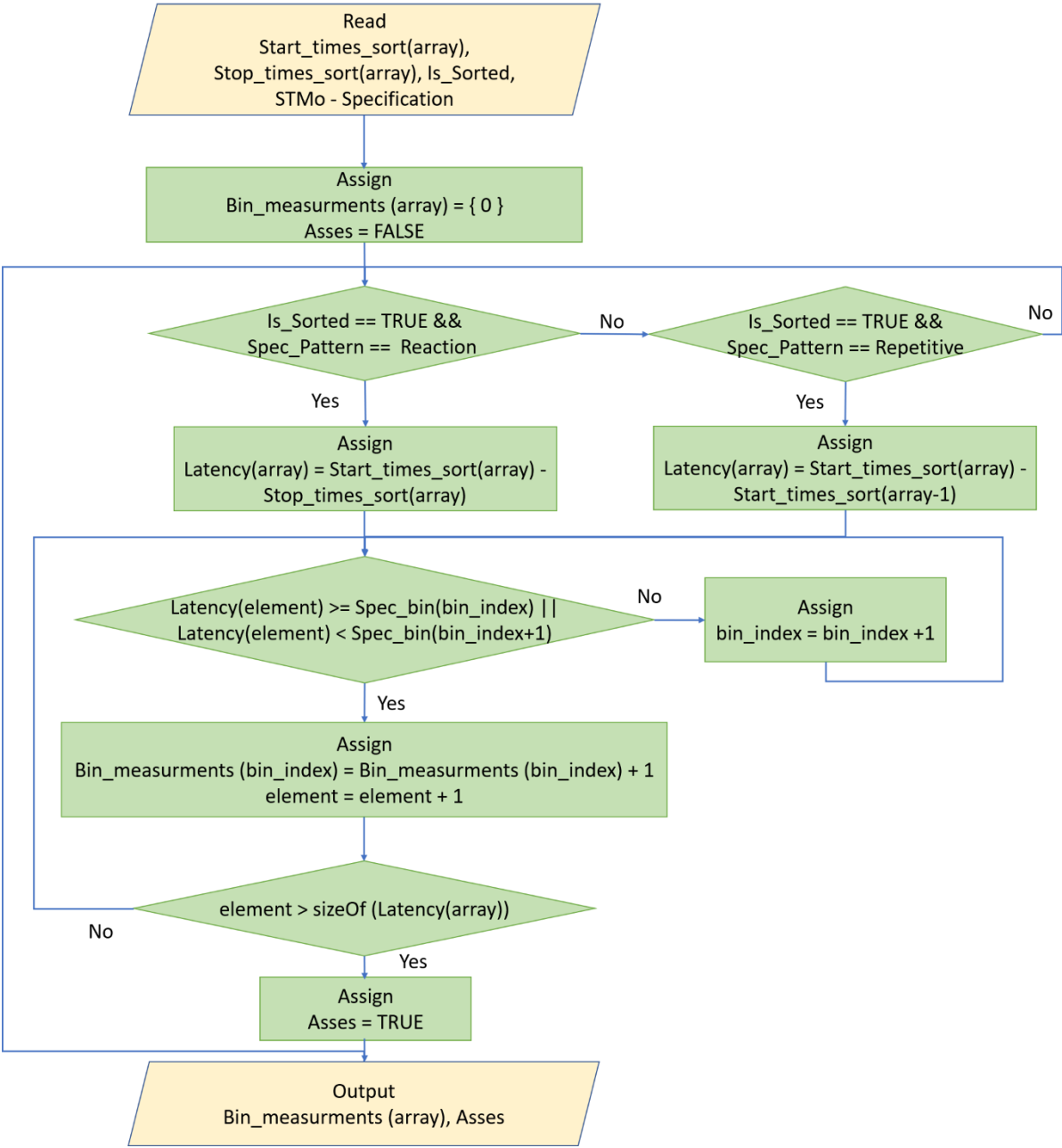


Figure 17: Flowchart for the implementation of Measurement module

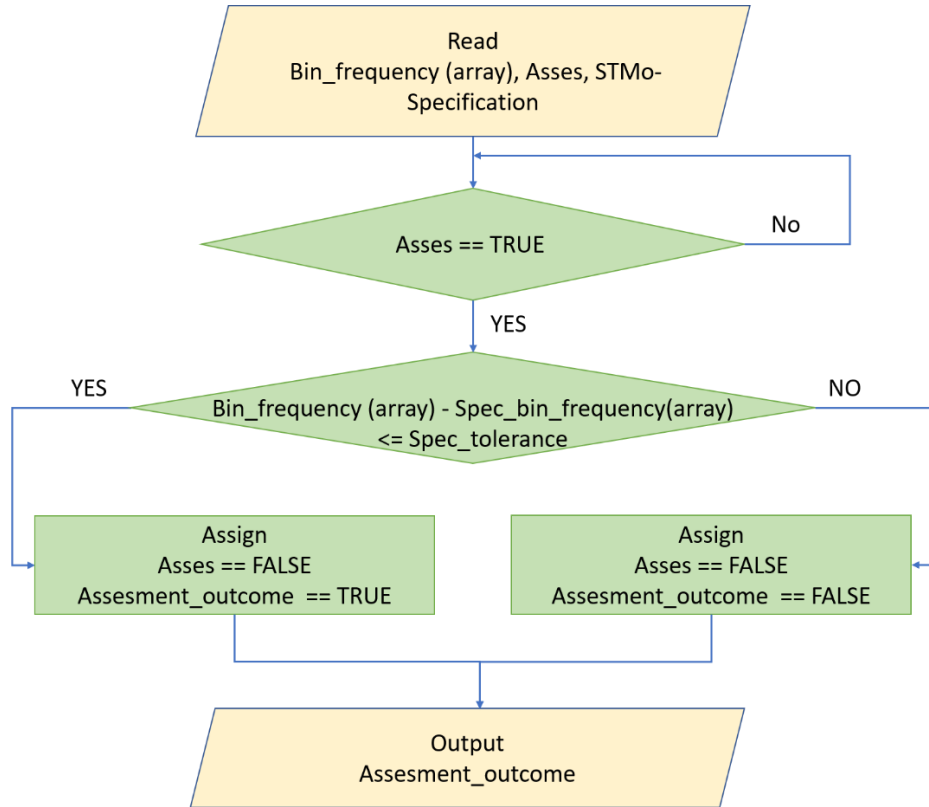


Figure 18: Flowchart for Assessment module implementation

Figure 18 illustrates the components of the STMo-system that have been implemented in VHDL. For both the Observer and Monitor, a dedicated top-level module is designed to instantiate the required submodules in accordance with the system specification. Detailed information regarding the automated generation of hardware modules based on the STMo-specification is provided in Section Teilbeitrag: B2.3.15.

Integration of the STMo-system with the processor is accomplished using the Block Design methodology in Xilinx Vivado. This approach facilitates modular and convenient integration. Figure 20: Block design of the STMo-system in Vivado presents the complete block design of the STMo-system as used in the implementation.

Teilbeitrag: B2.3.15 Synthesis concept for monitors

Partnerbeitragsbeschreibung

While the monitors defined in B2.3.12 and the observers defined in B2.3.13 were implemented manually in the form of prototypical hardware circuits in B2.3.14, a concept for the automatable synthesis of the monitor circuits from a model specification is to be developed in this partner contribution.

Ergebnisse

This section presents the concept of automatic synthesis of the Statistic-Timing Monitor (STMo) from a specification expressed in the STMo-Specification Language (STMoSL). This approach enables the automated generation of hardware circuits tailored to diverse specifications, streamlining the STMo development process.

The synthesis process, as depicted in Figure 21, begins with the STMo specification, which defines the timing behaviour of the firmware. This specification includes key parameters such as the number of distribution bins, window size, jitter distribution, and other details, as outlined in section Timing Specification. The STMoSL parser processes this specification and translates it into an intermediate data structure, making it suitable for further synthesis. This step converts a human-readable specification into a format that the synthesizer can interpret.

The STMoSL synthesizer consists of a set of Tcl scripts executed within the Vivado Tcl console. In addition to the parsed specification, the synthesizer requires configuration data from the firmware developer/user, which includes clock frequency, monitor and filter placement, communication protocols, and other system-specific parameters. Using this information, along with the parsed specification, the STMoSL synthesizer constructs the STMo system by using RTL components from the STMo Specification (STMoS) Hardware Library. The resulting STMo system is subsequently integrated with the RISC-V system-on-chip (SoC).

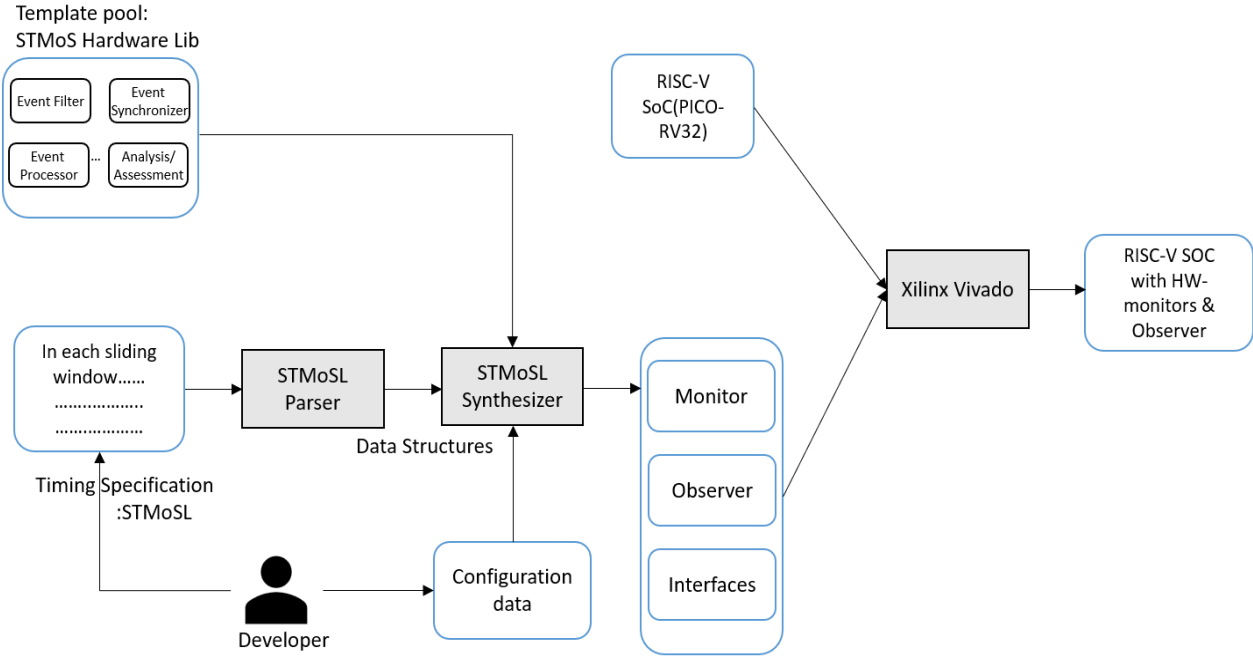


Figure 21: Overview of the STMo synthesis concept

Description of the STMo synthesis components

The following section explain the process of STMo synthesis by elaborating on each of the components involved in the synthesis.

a. Timing Specification in STMoSL

The execution-timing behaviour of the firmware is provided to STMo as a specification, which is expressed using Statistic-time monitoring specification language (STMoSL). This specification includes information such as window type, window size, tolerance, offset and jitter, using which the STMo can obtain the execution-time distribution of the firmware. As an example, Figure 22: Specification expressed in STMoSL shows the specification in STMoSL for the firmware provided by IMMS, (Institut für Mikroelektronik- und Mechatronik-Systeme) developed for the ASIC to integrate with the robotic demonstrator. Further details on STMoSL are provided in section Timing Specification.

```

In each sliding window of length 30
with tolerance 0% and arbitrary cold-start :

RISCV.GPIO.port[0]- port[7].0b00000001
RISCV.GPIO.port[0]- port[7].0b00000010

occurs within 5.52ms with offset 0ms and sym. jitter-distribution of {33.33% in
[-0.5ms to -0.2ms] , 33.33% [-0.2ms to 0.2ms], 33.33% [0.2ms to 0.5ms] }

```

Figure 22: Specification expressed in STMoSL

b. Template Pool: STMoS Hardware Library

STMo hardware library, also referred as a Template Pool, is a collection of RTL components which are the building blocks of STMo. The library includes various components such as event filter, sensor/comparator, observer and interface, Timed event generator and several others. Figure 19 shows the list of the RTL components that are available in the library pool. Further details on the functionality of these components are described in section Teilbeitrag: B2.3.14 Hardware components for monitor implementation. With the help of VHDL-2008 generics package, these components are made parameterizable so that the RTL components can be configured as per the STMo-specification. The generics package can be split into two parts: a header that declares the generic values and a body that defines the values (as illustrated in Figure 23 and Figure 24). The VHDL generate syntax was utilized to build the STMo with only the components that are required for the given specification and configuration data and thus optimizing the resource utilization. An example of generate statement is illustrated in Figure 25.

```

package spec_defined is new work.spec
generic map (
    -----Specification Generic Values-----
    window          => ("11"),
    pattern          => ("11"),
    max_bin_monitor => (5,5),
    totalevent_monitor => (10,10),
    time_spec       => (2000,2000),
    gap_spec        => ((-1000,-500,-499,-100,-99,10),
    startevent_spec => (1,3),
    stopevent_spec  => (2,4),
    bin_spectimeMat => ((2,4,1,2,1), (1,1,5,2,1)),
    tolerance       => (2,2),

```

Figure 23: Generic package Header

```

package spec is
Generic (
    -----Specification Generic-----
    window          : STD_LOGIC_VECTO
    pattern          : STD_LOGIC_VECTO
    max_bin_monitor : IntegerArray;
    totalevent_monitor : IntegerArray;
    time_spec       : IntegerArray;
    startevent_spec : IntegerArray;
    stopevent_spec  : IntegerArray;
    tolerance       : IntegerArray ;
    bin_spectimeMat : IntegerMatrix;

```

Figure 24: Generic package body

```

mon_int_gen :case comm_out_prot generate

    when 1 =>
        none_int : none_interface
        port map(
            timestamp_in    => timestamp_in,
            events_in       => events_in,
            timestamp_out   => timestamp,
            events_out      => events
        );

    when 3 =>
        uart_int : UART_receiver
        port map(
            clk             => clk,
            resetn         => resetn,
            events         => events,
            timestamp      => timestamp,
            uart_rx        => input
        );

```

Figure 25: Example of a generate statement

c. Configuration data

The configuration data, provided by the firmware developer/user, allows the STMo-system to seamlessly integrate with system on which the firmware is executed. In addition to this, it also provides necessary information for the synthesis of the Observer and the Monitor. The parameters in the configuration data along with their significance are described below.

- COMM_OB: Defines the communication protocol to be used for the observer interface.
- EVEN_SI: Event data width in bits.
- FILT_OBS: Indicates the placement of the event filter. Set to TRUE if filter is placed in observer otherwise FALSE
- FILT_MON: Indicates the placement of the event filter. Set to TRUE if filter is placed in Monitor Otherwise FALSE
- MON_PLA: Whether the monitor resides on the same chip as the observer or on a different chip. TRUE value indicates Monitor on the same chip otherwise FALSE
- COMM_OB_MO: Communication protocol to be used for the observer-monitor interface.

Along with the above information, the configuration data contains information necessary for the implementation of the communication protocols. Figure 26 shows an example of the configuration data used for the STMo-system generation and the values used for different communication protocols in the configuration data are as given below.

- 1: I/O connection- this protocol is used only when the monitor and the observer are on the same board and can communicate using inputs and outputs of the modules.
- 2: GPIO- this protocol can only be used for COMM_OB for now.
- 3: UART- this protocol can only be used for COMM_OB_MO and permit a serial transmission between the monitor and the observer.
- 4: I2C- Not yet implemented.
- 5: CAN- Not yet implemented.

```

-----Configuration part / Communication protocol-----
-----
filt_obs      => false,
filt_mon      => true,
even_si       => 8
comm_ob       => 2,
comm_ob_mo    => 1,
MON_PLA       => true,
-----UART CONF-----
CLK_FREQ      => 50e6,  -- system clock frequency in Hz
BAUD_RATE     => 115200, -- baud rate value
PARITY_BIT    => "none", -- type of parity: "none", "even", "odd", "mark", "space"
USE_DEBOUNCER => True,  -- enable/disable debouncer
memory_size   => 4

```

Figure 26: Illustration of the configuration data

d. STMoSL Parser

The STMoSL Parser is a small Java software to read text-files with well-formed, formal specifications in STMoSL and to translate them from their human-readable, pattern-based format into an intermediate representation for the synthesis process. Then, to avoid the generation of inconsistent monitors, a semantic analysis of the specified timing properties is performed. As its output the Parser generates an intermediate representation that can be used by different synthesizer backends to allow for a synthesis into both hardware-based STMos as well as different executables for software-based STMos.

e. STMoSL Synthesizer

The STMoSL synthesizer utilizes the parsed STMo specification and configuration data to generate a customized, hardware-based STMo system. This synthesis process is automated using a set of Tcl scripts executed within Vivado's Tcl console.

The first script is responsible for generating a VHDL specification component. It navigates to the target directory, removes any existing version of the VHDL specification file to ensure a clean start, and then reads the parsed specification data from the STMoSL parser along with the configuration data. Using this information, it creates a new VHDL specification component. Additionally, the script executes commands to upgrade the file to VHDL-2008, enabling advanced language features, and assign it to a dedicated library for better project organization.

The second Tcl script updates the design hierarchy and generates the required RTL components by instantiating elements from the template pool according to the given specification. Once synthesized, the monitoring system is integrated with the RISC-V SoC, followed by bitstream generation and the creation of hardware hand-off files for FPGA implementation. Figure 34 shows hardware utilization of the complete system on the FPGA and Figure 33 shows the block-design snapshot of STMo's integration with the RISC-V core.

I.2.3 AP3: Anforderungen an Vertrauenswürdigkeit von IP und Designflow

Teilbeitrag: B3.2.10 Semi-formal verification of runtime monitors for processor components

Partnerbeitragsbeschreibung

This partner contribution examines how the monitors from B2.3.14 can be integrated into verification methods. For simulative verification approaches, suitable simulation models of the Monitors are being researched and developed and test approaches (stimulus generation) investigated. The aim is to verify the correct functioning of the monitors and to test the effectiveness of monitors against the intended class of attacks.

Ergebnisse

The concept for verifying the functionality of the run-time monitor through semi-formal verification method is elaborated in the section Teilbeitrag: B1.3.6: Verification concept for observers and runtime monitors. The implementation of the three different verification approaches at design time and run-time is explained in detail in this section.

a. Design-time Verification - with testbenches

In this approach, to validate the functionality of the Statistic-Timing Monitor (STMo), a set of parameterized test benches was developed to simulate the RTL components of the monitoring system. These test benches are designed to emulate the event generation process of the firmware and replicate its execution timing behaviour through VHDL wait statements.

A sample test bench, illustrated in Figure 27 generates timing behaviour composed of three distinct execution bins. Further details on timing-behaviour, execution bins and specification are documented in the section Teilbeitrag: B2.3.12 Monitor modelling.

To verify the functionality of the STMo-system, test benches are designed to align with the timing behaviour of the firmware, which is provided in the STMo-specification. These test benches are then utilized for the simulation of the STMo components. Figure 29 shows the waveforms, presenting key signals necessary for verification. Each signal and its significance are explained below.

- Clk_tb: Indicates the global clock signal, which drives the timing for the entire system.
- Resetn_tb: Indicates active low reset signal which means the systems is held at reset when the signal is at logic 0.
- Events: An 8-bits signal that indicates the events generated by the testbench which is of STD_LOGIC_VECTOR type.
- Spec_bin_frequency: A matrix of integers specifying the expected jitter count per bin, as defined in the STMo-specification.
- Measured_bin_frequency: Holds the measured bin counts computed by the STMo during runtime.
- Current_window_size: Indicates the current number of measurements within the ongoing monitoring window.
- Spec_window_size: This signal indicates the window length parameter provided in the specification.
- Verdict_tb: A STD_LOGIC signal that outputs the final decision made by the STMo system.

The firmware is then compiled and executed on a RISC-V core implemented on a PYNQ-Z2 FPGA board, where the STMo is also interfaced via GPIO. To enable real-time signal visualization, an Internal Logic Analyzer (ILA) IP is integrated into the design. The ILA captures data for selected signals when a predefined trigger condition is met. Figure 31 presents the Verdict and execution_time signal data recorded during the STMo's operation.

In this work, the real-time monitor output is validated by comparing it with pre-recorded simulation results at run-time. For that, a wrapper module is implemented to store simulation results at design time, before it is then used for comparison against the real-time outcomes of the STMo at runtime. This approach facilitates the verification of the STMo across various specifications with minimal manual effort.

```
int main(int argc, char ** argv){
    unsigned int * arr = (unsigned int *)argv[1];
    for (int i=0;i<=17;i++){
        reg_outp = 1;
        __asm__ ("nop");
        __asm__ ("nop");
        __asm__ ("nop");
        reg_outp = 2;

        __asm__ ("nop");

        reg_outp = 1;
        __asm__ ("nop");
        __asm__ ("nop");
        __asm__ ("nop");
        __asm__ ("nop");
        reg_outp = 2;
    }
}
```

Figure 30: Firmware with NOPs

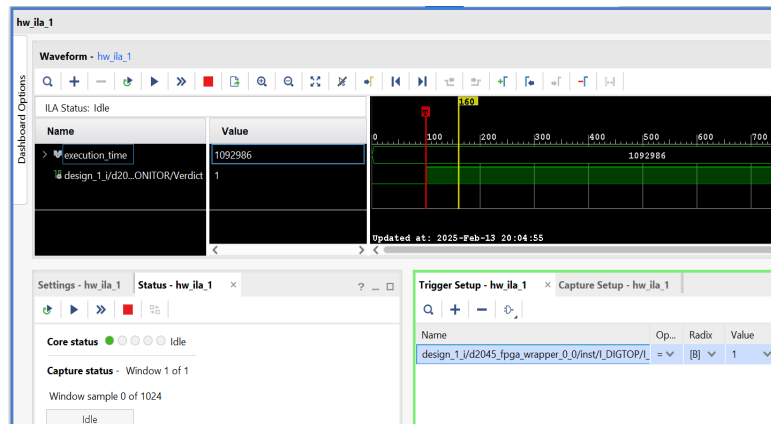


Figure 31: Real-time data using ILA

c. Run-time Verification- in comparison with software Monitor outcome

Another part of the run-time verification concept of this work follows a similar equivalence checking approach as the previous one, with the key distinction that the Statistic-Timing Monitor (STMo) results are not validated against the outcomes of a simulation but against the outcomes of a software-based STMo system corresponding to the same specification as the hardware-based implementation. That means, the software-monitor system is expected to replicate the functionality of the hardware STMo, and to produce concurrent results for a given specification for both monitoring systems.

Figure 32 illustrates the architecture for this approach. The STMo is implemented on the programmable logic (PL) section of the FPGA, while its software counterpart executes on the processing-system (PS) of the ZYNQ-7000 SoC, which is an ARM-based processor. To verify result consistency, an equivalence check module is employed on the PS to compare the outputs of both monitoring systems.

However, at the current stage, only the hardware-based STMo system has been successfully implemented on the programmable logic, while the software-based monitoring system is currently still under development. The proposed verification approach - comparing the real-time outputs of the STMo and the software monitors - will be carried out as part of future work.

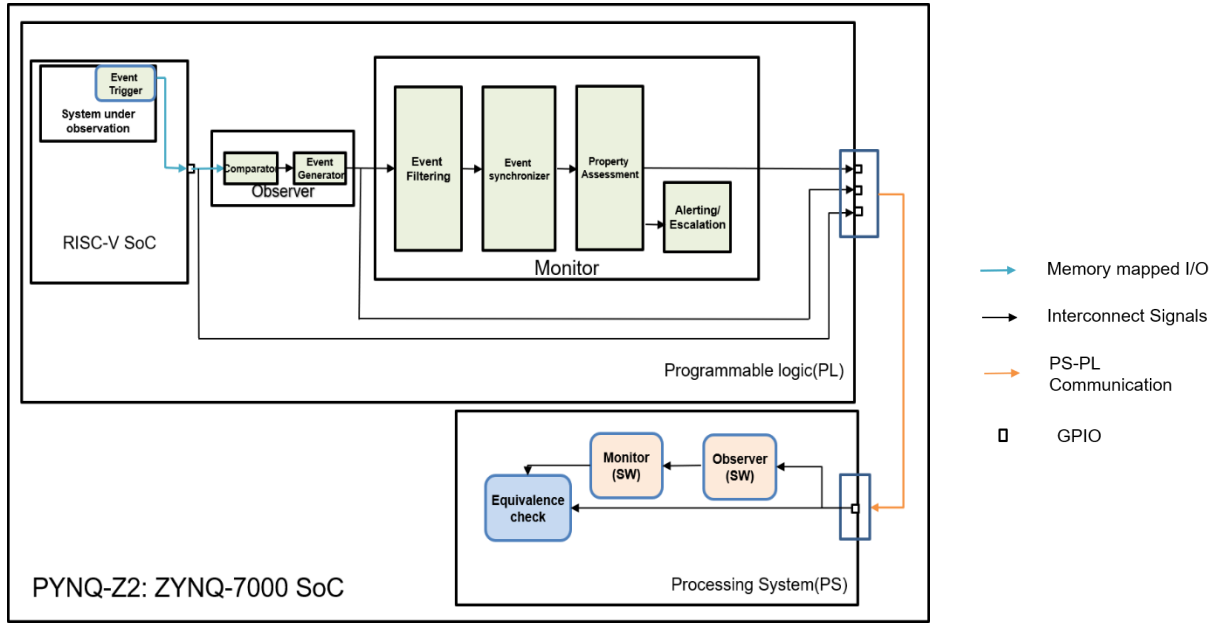


Figure 32: Run-time verification using software-Monitor system

I.2.4 AP4: Anforderungen an Vertrauenswürdigkeit von IP und Designflow

Teilbeitrag: B4.1.8 Prototypical integration and evaluation of runtime monitors for a RISC-V processor

Partnerbeitragsbeschreibung

This partner contribution demonstrates the runtime monitors developed in B2.3.14 for processors in a virtual platform and possibly an FPGA demonstrator (Xilinx Zynq-7000 or Zynq UltraScale+) of the (RISC-V) TEE architecture.

Ergebnisse

This section presents the integration and evaluation of the STMo-system on two different hardware platforms. The functionality of the STMo-system is assessed using firmware that exhibits distinct execution timing behaviour.

The Statistic-Timing Monitor (STMo) system, consisting of the Observer and the Monitor, is deployed on hardware platforms for real-time evaluation. The integration of STMo is carried out on two distinct Hardware platforms. The first platform is a RISC-V-based System-on-Chip (SoC), also referred as PICO RISC-V, implemented on a PYNQ-Z2 FPGA. The initial evaluation was performed on this platform with generic firmwares to verify the run-time functionality of STMo with various specification configurations. The second platform is a demonstrator provided by IMMS (Institut für Mikroelektronik- und Mechatronik-Systeme), which also incorporates a RISC-V core. With this platform, STMo is evaluated with a real-time firmware. Further details on the integration of the STMo with these platforms are presented in the coming section.

Another point to add here is the additional efforts spent on setting-up both the evaluation platforms, which extended beyond the initial scope. Key tasks such as compiler tool chain installation on processing-system (PS), addition of another CAN node for data receipt, Development of domain-specific-language for monitor modelling required more efforts than anticipated in the beginning.

Integration with PICO RISC-V on PYNQ-Z2 Board

This section provides details on the implementation of the STMo on Xilinx PYNQ-Z2 board that sources ZYNQ SoC. To facilitate the execution of firmware requiring monitoring, the work incorporates an open-source RISC-V based SoC project, sourced from the GitHub repository "RISC-V Integration for PYNQ" ([GitHub - drichmond/RISC-V-On-PYNQ: RISC-V Integration for PYNQ](https://github.com/drichmond/RISC-V-On-PYNQ)). To establish the connection between the RISC-V SoC and the STMo-system, the PICO RISC-V is further developed to include a General-Purpose Input/output (GPIO) module that provides a 32-bit GPIO interface. With this enhancement, it is possible to send distinct events which can then be forwarded to the monitoring system for analysis. Figure 33 shows the block design of the Vivado project, incorporating the enhanced RISC-V SoC along with the STMo-system.

To compile firmware for the RISC-V core, the GNU toolchain for a bare-metal RISC-V architecture was installed on the Processing System (PS) present within the ZYNQ SoC. This setup enables the smooth compilation and execution of applications directly on the hardware. The detailed description of the toolchain, along with its installation process and usage details is provided here- [Github-Compiling the RISC-V GCC Toolchain](#).

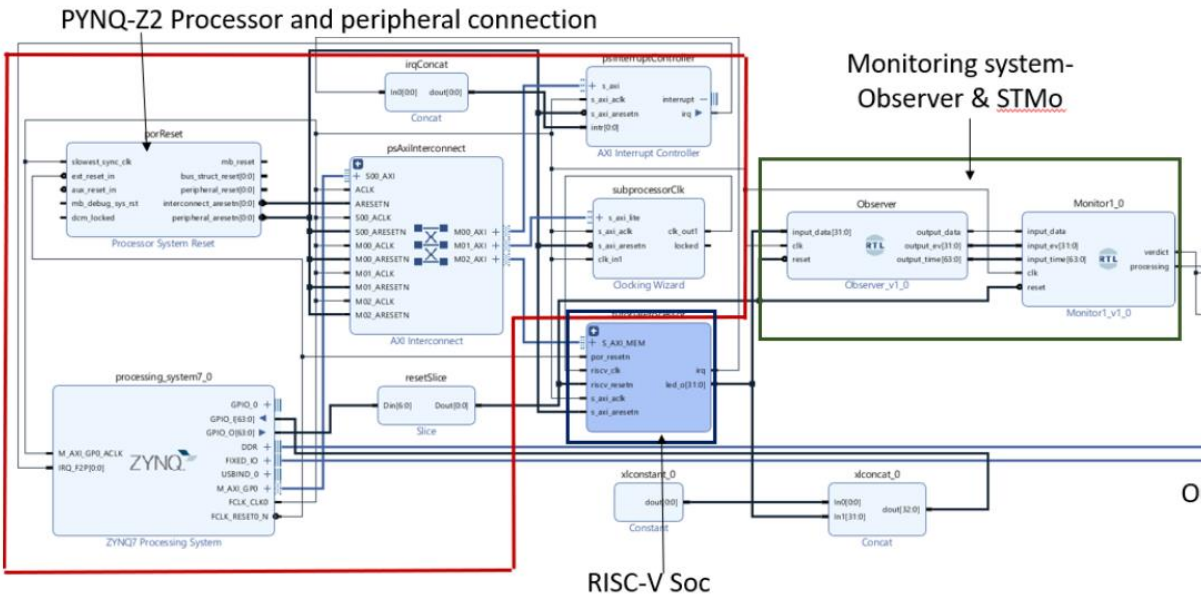


Figure 33: STMo integration with PICO RISC-V

The synthesis process which is explained in section STMoSL Synthesizer is utilized to automatically generate the STMo-system. Figure 34 presents the synthesis and implementation results for the monitoring system integrated with the RISC-V SoC. The figure on the left illustrates the synthesis results, which detail the overall resource utilization of the design. This includes metrics such as the usage of Slice LUTs (Look-Up Tables), Slice Registers, LUTs configured as Logic, LUTs configured as Memory, Multiplexers and Block RAM Tiles. The figure on the right displays the floorplan view of the design after the implementation process, specifically targeting the Pynq-Z2 board. The floorplan view provides a visual representation of the physical layout of the design on the FPGA, with color-coded regions representing various components, as given below.

- PYNQ-Z2 Processor
- RISC-V Processor
- Monitor
- PYNQ-Z2 System
- ILA System
- Observer

Name	Slice LUTs (53200)	Slice Registers (106400)	LUT as Logic (53200)	LUT as Memory (17400)	Slice (13300)	F7 Muxes (26600)	Block R# Tile (14)
tutorial_wrapper	10338	10606	9619	719	4270	373	27
> dbg_hub (dbg_hub)	475		451	24	226	0	0
> tutorial_1 (tutorial_1)	8037		7857	180	3146	281	16
irqConcat (tutorial_1_irqConcat)	0		0	0	0	0	0
> Monitor1_0 (tutorial_1_Monitor1_0)	4099		4099	0	1404	128	0
> inst (tutorial_1_Monitor1_0_inst)	3976		3976	0	1374	128	0
> mon (tutorial_1_Monitor1_0_inst_mon)	3976		3976	0	1374	128	0
> assess (tutorial_1_Monitor1_0_inst_mon_assess)	130		130	0	45	0	0
> flit_on_m... (tutorial_1_Monitor1_0_inst_mon_assess_flit_on_m...)	66		66	0	48	0	0
> flit_on_m... (tutorial_1_Monitor1_0_inst_mon_assess_flit_on_m...)	1515		1515	0	606	128	0
> valid (tutorial_1_Monitor1_0_inst_mon_assess_flit_on_m..._valid)	2323		2323	0	728	0	0
> Observer (tutorial_1_Observer)	29		29	0	47	0	0
> porReset (tutorial_1_porReset)	16		15	1	11	0	0
> processing_system... (tutorial_1_processing_system...)	0		0	0	0	0	0
> psAxilinterconnect (... (tutorial_1_psAxilinterconnect (...)	1392		1270	122	634	0	0
> psInterruptControll... (tutorial_1_psInterruptControll...)	70		70	0	26	0	0
> resetSlice (tutorial_1_resetSlice)	0		0	0	0	0	0
> subprocessorCik (t... (tutorial_1_subprocessorCik (t...)	1104		1104	0	643	153	0
> tutorialProcessor (t... (tutorial_1_tutorialProcessor (t...)	1323		1266	57	460	0	16
> verdict_lightning_0... (tutorial_1_verdict_lightning_0...)	4		4	0	2	0	0
> xiconcat_0 (tutorial_1_xiconcat_0)	0		0	0	0	0	0
> xiconstant_0 (tutori... (tutorial_1_xiconstant_0 (tutori...)	0		0	0	0	0	0
> u_ila_0 (u_ila_0)	1826		1311	515	1049	92	11

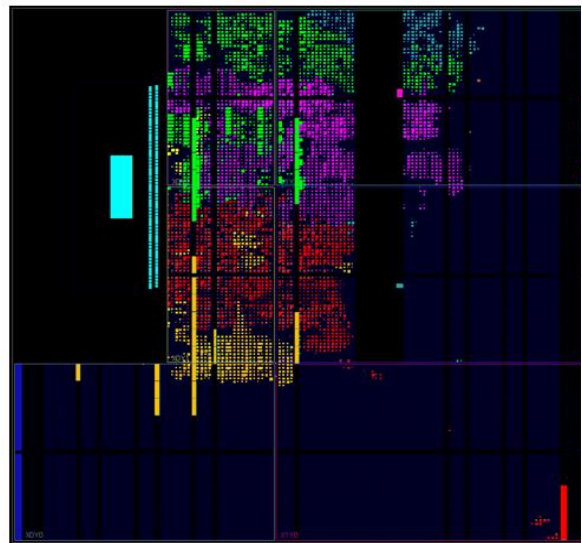


Figure 34: Synthesis and implementation result

The synthesized bitstream of the STMo-system was implemented on a PYNQ-Z2 board using the overlay approach. Using Overlay, it's possible to interact with the programmable logic (PL) of the PYNQ board. The Overlay defines the hardware configuration of the FPGA, including custom logic blocks, IP cores, and data paths, all of which can be accessed and controlled through Python using the PYNQ framework. This method provides a flexible and efficient way to deploy and test RTL designs directly on the FPGA fabric without requiring low-level hardware programming.

Figure 35 illustrates the Python script used for the overlay process. The script loads the bitstream file (tutorial.bit)—which encapsulates the hardware design of the STMo-system—onto the FPGA. Once the bitstream is successfully loaded, the system is accessible via the TutorialOverlay class, which acts as a Python wrapper for interacting with the FPGA hardware described by the bitstream.

```
import sys
import os
os.chdir("/home/xilinx/RISC-V-On-PYNQ/riscvonpynq/picorv32/tut")
sys.path.insert(0, '/home/xilinx/RISC-V-On-PYNQ/riscvonpynq/picorv32')
sys.path.insert(0, '/home/xilinx/RISC-V-On-PYNQ')
from tut.tutorial import TutorialOverlay
overlay = TutorialOverlay("tutorial.bit")
```

Figure 35: Python Libraries used for Overlay

Integration with IMMS Demonstrator

The synthesized Statistic-Timing Monitor (STMo) system was integrated with the demonstrator provided by IMMS. This demonstrator implements the ASIC D2045 design on an FPGA, complemented by a Raspberry Pi board. The design features a RISC-V-based CPU core, an on-chip FPGA area serving as a Trusted Execution Environment (TEE), a CAN bus controller, and CAN driver firmware responsible for managing interactions with the CAN controller.

To facilitate the integration of the STMo system with the demonstrator, an 8-bit RISC-V GPIO interface is utilized, which is configurable through the crossbar. This connection enables the STMo system to access essential data from the RISC-V core. Figure 36 illustrates the integration of STMo with the demonstrator. Since the STMo components were manually added to the design, they are not directly visible in the block design. Instead, only the top module- D2045 FPGA Wrapper- is displayed (Figure 36, left). However, the right side of Figure 36 presents a detailed list of source files within the top module, where the individual STMo components can be identified.

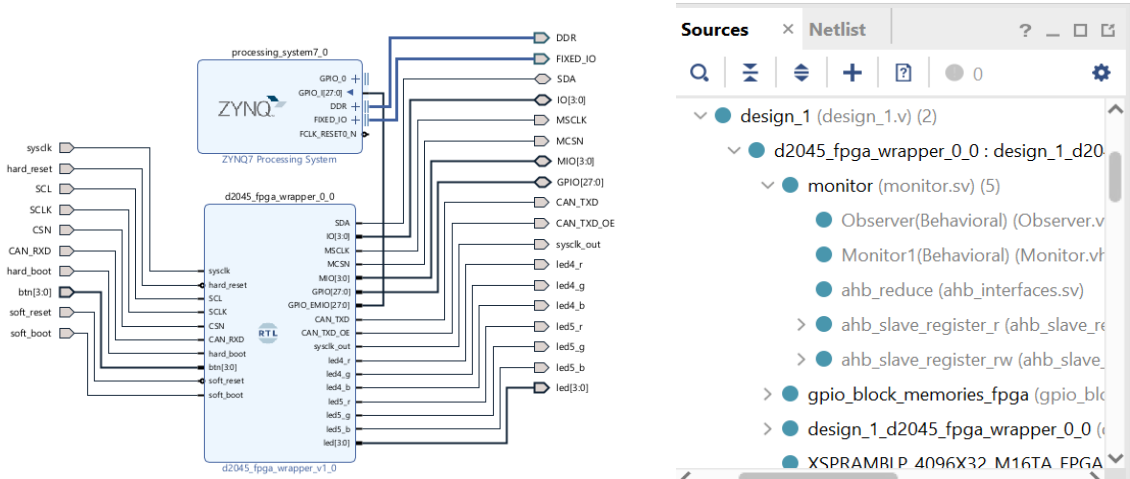


Figure 36: STMo integration with IMMS Demonstrator

The bitstream and hardware handoff files are generated for the integrated design and subsequently implemented on the demonstrator. The floorplan view in Figure 37 offers a visual representation of the design's physical layout on the FPGA, with the tiles corresponding to the STMo components - Observer and Monitor - highlighted in different colours for distinction.

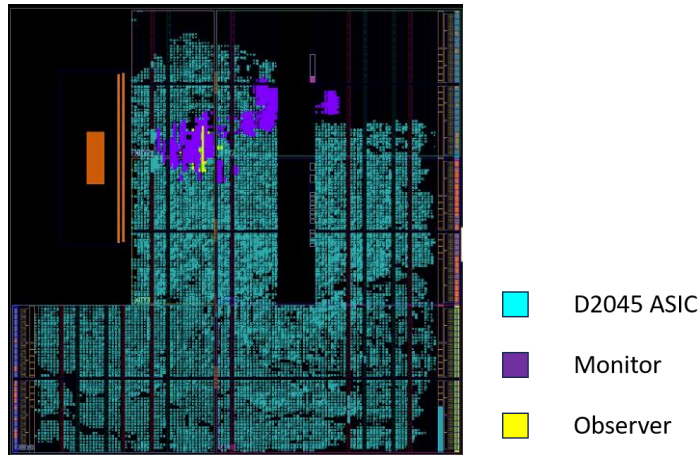


Figure 37: Floor plan of the IMMS demonstrator integrated with STMo

Evaluation on PICO RISC-V

After integrating the STMo with the RISC-V SoC on the hardware, the next step involves executing the firmware whose timing behaviour needs to be monitored. This execution is performed on the RISC-V core using the Python Notebook, which provides a web-based interface for interact with the FPGA. The Python Notebook enables users to load the bitstream, control hardware peripherals, and interface with the FPGA fabric through Python scripts.

To facilitate monitoring, the firmware is modified to include event annotations, allowing the RISC-V core to transmit event data to the STMo. Using this event data, the STMo determines the execution-time distribution of the firmware. Figure 38 illustrates these event annotations, while further details on execution-time distribution are available in deliverable D10-2A.

Various firmware was developed to evaluate the functionality of the STMo. In the beginning the firmware was developed using NOP (No Operation) instructions, as their execution time is precisely known and remains consistent across different processor conditions. This approach facilitates the development of firmware that aligns with the specification, thereby reducing the time required to create firmware for various verification scenarios. The left side of Figure 38 shows a firmware snippet with NOPs generated for a specification. Later, the NOPs were replaced with simple instructions (Figure 38 right) and eventually with simpler C programs to evaluate the STMo functionality. The various firmware developed, helped to evaluate the STMo functionalities for different specifications including reaction and repetitive patterns in conjunction with parameters such as window size, tolerance, and distribution.

To evaluate real-time signals of STMo, an Internal Logic Analyzer (ILA) IP is integrated into the design to provide visualization of the signals. The ILA captures data for selected signals when a predefined trigger condition is met enabling the verification of real-time signal values.

The work also includes evaluation of the real-time STMo output by comparing it with pre-recorded simulation results at run-time. For that, a wrapper module is implemented to store simulation results at design time, before, it is then used for comparison against the real-time outcomes of the STMo at runtime. This approach facilitated the evaluation of the STMo across various specifications/firmware with minimal manual effort.

```

int main(int argc, char ** argv){
  unsigned int * arr = (unsigned int *)argv[1];
  for (int i=0;i<=17;i++){
    reg_outp = 1;
    __asm__ ("nop");
    __asm__ ("nop");
    __asm__ ("nop");
    reg_outp = 2;

    __asm__ ("nop");

    reg_outp = 1;
    __asm__ ("nop");
    __asm__ ("nop");
    __asm__ ("nop");
    __asm__ ("nop");
    reg_outp = 2;
  }
}

reg_outp = 1;
for(int k = 0;k<=100;k++){
  __asm__ volatile (
    "add %[a], %[a], %[b]\n"
    : [a] "+r" (a)
    : [b] "r" (b)
  );
  reg_outp = 2;

  reg_outp = 1;
  for(int k = 0;k<=105;k++){
    __asm__ volatile (
      "mul %[a], %[a], %[factor]\n"
      : [a] "+r" (a)
      : [factor] "r" (factor)
    );

    __asm__ volatile (
      "add %[a], %[a], %[constant]\n"
      : [a] "+r" (a)
      : [constant] "r" (constant)
    );
  }
  reg_outp = 2;
}

```

Figure 38: Firmware examples used for evaluation on PYNQ-Z2 board

a. Evaluation on IMMS demonstrator

To evaluate the functionality of STMo on the IMMS demonstrator, a firmware developed by IMMS for the ASIC's integration with the Siemens robotic demonstrator was utilized. This firmware is responsible for retrieving sensor data and transmitting it to the CAN controller. However, unlike the ASIC, the demonstrator did not have an analogue front-end for direct sensor interfacing. Therefore, the firmware was modified to internally generate data before sending it to the CAN controller.

Initially, the CAN firmware was subjected to execution-timing measurements, revealing a constant execution time of 54.65 ms. This consistent execution time resulted as the CAN network is having a single CAN node to transmit the data. Figure 39 (right) presents the specification derived from this measurement, expressed in STMo Specification Language (STMoSL), with a single execution bin. Further details on STMoSL and execution bins can be found in the section Teilbeitrag: B2.3.12 Monitor modelling. The STMo was synthesized based on this specification and evaluated using the CAN firmware.

Figure 40 illustrates the real-time CAN verdict output signal, captured using ILA, which indicates whether the firmware adheres to the specified timing behaviour. A TRUE verdict signifies compliance with the timing specification, while a FALSE verdict indicates deviations. The results demonstrated that STMo consistently produced a TRUE verdict for the considered CAN firmware and a FALSE verdict when this firmware was modified to alter its timing behaviour.

<p>In each sliding window of length 100 with tolerance 0% and arbitrary cold-start :</p> <p>RISCV.GPIO.port[0]- port[7].0b00000001 RISCV.GPIO.port[0]- port[7].0b00000010</p> <p>occurs within 54.65ms with offset 0ms and jitter-distribution of {100% in [0ms]}</p>	<p>In each sliding window of length 30 with tolerance 0% and arbitrary cold-start :</p> <p>RISCV.GPIO.port[0]- port[7].0b00000001 RISCV.GPIO.port[0]- port[7].0b00000010</p> <p>occurs within 5.52ms with offset 0ms and sym. jitter-distribution of {33.33% in [-0.5ms to -0.2ms] , 33.33% [-0.2ms to 0.2ms], 33.33% [0.2ms to 0.5ms] }</p>
--	--

Figure 39: STMoSL specification with single and multiple execution bins.

To further evaluate STMo in a multi-bin configuration, the firmware was modified to exhibit varying execution times, distributed across three different bins. This was achieved by adjusting the data length transmitted over the CAN network. Figure 39 (right) shows the corresponding specification for this modified firmware. The STMo was generated accordingly and evaluated for both positive and negative verdicts, validating its ability to monitor diverse execution-time distributions.

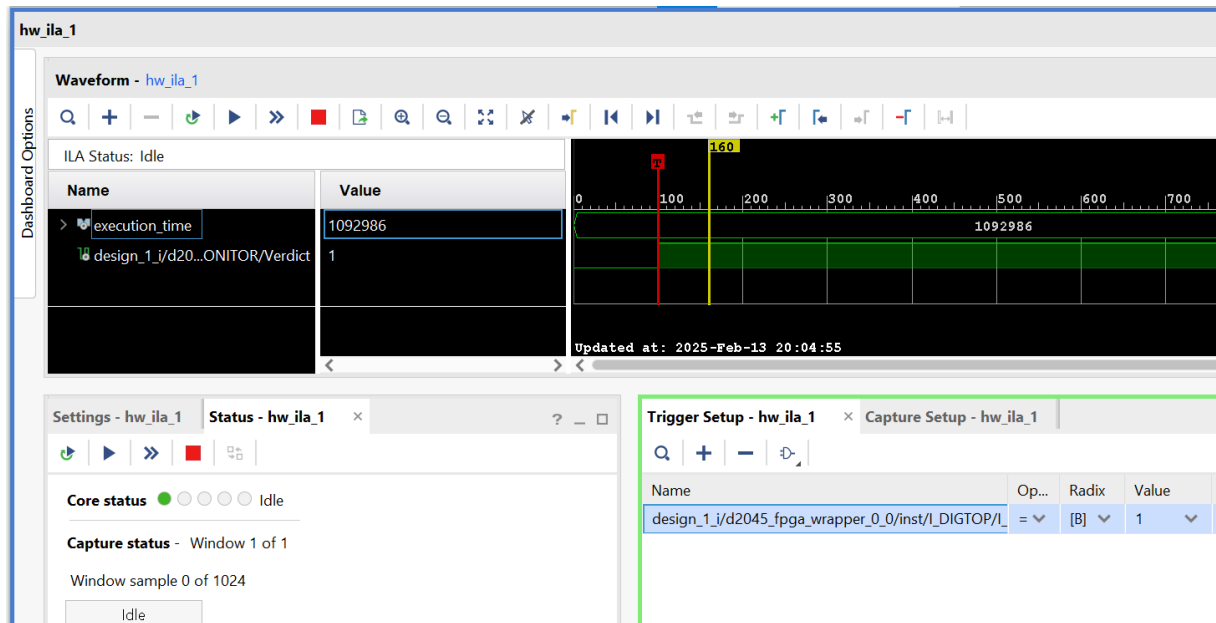


Figure 40: Real-time visualization of STMo verdict using ILA

I.3 Voraussichtlicher Nutzen, Verwertbarkeit der Ergebnisse

I.3.1 Nutzen und Verwertbarkeit der Ergebnisse aus Arbeitspaket 1

I.3.1.1 Nutzen und Verwertbarkeit Beitrag B1.3.6: Verification concept for observers and runtime monitors

Ziel des Beitrages B1.3.6

This partner contribution defines the concept for the verification of observers and runtime monitors to be implemented in WP3 (B3.2.10). Based on the identified requirements, a concept is developed that is suitable for verifying the monitored properties and criteria.

Nutzen, Verwertbarkeit des Beitrags B1.3.6

Key benefit of this contribution is the development of a project-internal verification concept that compares the model-based, software-based and hardware-based monitoring-results to verify the correct synthesis of STMoSL-based monitor-specifications by equivalence checking between the three different solutions. The concept is to be implemented in contribution B3.2.10 and enables realtime-analysis and -verification of the Statistic-Timing Monitor (STMo) against their defined specifications. For the later implementation of software-based monitors, it allows for the semi-formal equivalence checking. Individually, it is not meant for external exploitation.

I.3.2 Nutzen und Verwertbarkeit der Ergebnisse aus Arbeitspaket 2

I.3.2.1 Nutzen und Verwertbarkeit Beitrag B2.3.12 Monitor modelling

Ziel des Beitrages B2.3.12

In this partner contribution, monitors are defined and modelled that can detect behavioural deviations in relation to a given specification. The exact properties and criteria to be monitored were defined in B1.2.11, the focus should be on functional behaviour, timing behaviour and power dissipation of RISC-V processors. This concept of monitors is to be considered as a general concept, i.e. that it should also be transferable to other processors and, in particular, it should be independent of an implementation in hardware or software at this point in time.

Nutzen, Verwertbarkeit des Beitrags B2.3.12

The results of contribution B2.3.12 significantly improve the usability of STMoSL-based monitors by providing a comprehensive model for design and implementation of the monitors capable of detecting execution timing deviations for embedded firmware. In terms of usability, the monitor modelling approach integrates seamlessly with observers via a configurable interface, supporting multiple communication protocols. This flexibility enables deployment in diverse hardware environments, whether the monitor is on-chip or external. The usage of the domain-specific specification language (STMoSL) allows developers to define the timing behavior formally and translate these requirements directly into the monitoring system. Language models will be further evaluated and will be considered to contribute to open-source software and further research projects.

I.3.2.2 Nutzen und Verwertbarkeit Beitrag B2.3.13 Observer definition

Ziel des Beitrages B2.3.13

In this partner contribution, observers are developed, i.e. hardware circuits that act as probes and collect the data required by the monitors (defined in B2.3.12) of the RISC-V processor to be monitored, making it analysable. If necessary, the observers must filter or average data in order to suppress measured values that are not required.

Nutzen, Verwertbarkeit des Beitrags B2.3.13

Contribution B2.3.13 provides the detailed design of observers that act as probes to collect event data required by monitors. In terms of usability, the observer design supports multiple communication protocols (e.g., GPIO, SPI, I2C, CAN), enabling flexible integration with various processor platforms and monitoring setups. The modular structure allows configuration based on specified system requirements. The resulting observer concept, ensuring that monitors receive accurate,

time-stamped, and relevant event data for precise evaluation of the timing behavior, is used for the integration of the STMo in the target systems (e.g. the demonstrators of the project) and will be jointly exploited together with the monitoring concept. An individual exploitation of the observer concept is not planned.

I.3.2.3 Nutzen und Verwertbarkeit Beitrag B2.3.14 Hardware components for monitor implementation

Ziel des Beitrages B2.3.14

In this partner contribution, the monitors that are defined and specified in B2.3.12 are designed as hardware circuits together with the connection to the observers that are defined in B2.3.13. These hardware circuits can be configurable, parameterizable or even fully programmable and are implemented as simulation components and prototype RTL components.

Nutzen, Verwertbarkeit des Beitrags B2.3.14

The results of contribution B2.3.14 include a set of hardware components for implementing monitors as defined in B2.3.12, integrated with observers from B2.3.13. A key advantage is that these components are configurable and parameterizable, allowing the monitoring system to be tailored to specific requirements without redesigning the entire hardware. The results can be used to build a customized monitoring solutions by reusing and configuring the hardware library components facilitating the seamless integration of runtime monitors into diverse environments. Because of the additional costs for hardware-based STMo systems there is currently no further exploitation planned. Nevertheless, the library components will be further evaluated and might be considered to contribute to an open-source software and further research projects.

I.3.2.4 Nutzen und Verwertbarkeit Beitrag B2.3.15 Synthesis concept for monitors

Ziel des Beitrages B2.3.135

While the monitors defined in B2.3.12 and the observers defined in B2.3.13 were implemented manually in the form of prototypical hardware circuits in B2.3.14, a concept for the automatable synthesis of the monitor circuits from a model specification is to be developed in this partner contribution.

Nutzen, Verwertbarkeit des Beitrags B2.3.15

The work presented in contribution B2.3.15 provides a synthesis concept enabling the automated generation of Statistic-Timing Monitors from formal specifications. Central to this approach is the direct translation of timing requirements, expressed in the Statistic-Timing Monitor Specification Language, into optimized hardware implementations. This automated workflow significantly reduces manual design effort, accelerates the overall development process, and circumvents manual faults. By utilizing a library of parameterizable RTL components, the synthesis process ensures that only the necessary modules are instantiated, leading to resource-efficient and specification-compliant monitor design providing a reliable runtime monitoring solution suitable for a wide range of embedded applications. Both, the synthesis concept and the developed synthesis process will be further evaluated and will be considered to contribute to open-source software and further research projects. Furthermore, it will be the base-line for the development of a corresponding synthesis process for software-based STMos.

I.3.3 Nutzen und Verwertbarkeit der Ergebnisse aus Arbeitspaket 3

I.3.3.1 Nutzen und Verwertbarkeit Beitrag B3.2.10: Semi-formal verification of runtime monitors for processor components

Ziel des Beitrages B3.2.10

This partner contribution examines how the monitors from B2.3.14 can be integrated into verification methods. For simulative verification approaches, suitable simulation models of the Monitors are being researched and developed and test approaches (stimulus generation) investigated. The

aim is to verify the correct functioning of the monitors and to test the effectiveness of monitors against the intended class of attacks.

Nutzen, Verwertbarkeit des Beitrags B3.2.10

Contribution B3.2.10 used the semi-formal verification methodology of B1.3.6 to ensure the correctness of the Statistic-Timing Monitors. While the contribution missed to implement the goal of checking the equivalence of the hardware-based STMo vs. a software-based equivalent, it implemented the equivalence-check towards a model-based simulation, enabling validation of monitors under realistic operating conditions by comparing the runtime outcomes against pre-recorded simulation data.

I.3.4 Nutzen und Verwertbarkeit der Ergebnisse aus Arbeitspaket 4

I.3.4.1 Nutzen und Verwertbarkeit Beitrag B4.1.8: Prototypical integration and evaluation of runtime monitors for a RISC-V processor

Ziel des Beitrages B4.1.8

This partner contribution demonstrates the runtime monitors developed in B2.3.14 for processors in a virtual platform and possibly an FPGA demonstrator (Xilinx Zynq-7000 or Zynq UltraScale+) of the (RISC-V) TEE architecture.

Nutzen, Verwertbarkeit des Beitrags B4.1.8

The results of contribution B4.1.8 offer important benefits by demonstrating the successful integration and evaluation of the Statistic-Timing Monitors on RISC-V processors. The key benefit is the proof of practical feasibility achieved by integrating the STMo system into two hardware platforms—a PYNQ-Z2 FPGA with a RISC-V SoC and the IMMS demonstrator with an ASIC-based RISC-V design. In both setups, the monitors reliably detected compliant and non-compliant timing behavior across different firmwares giving a proof of the overall observer-monitor-concept.

The usability of these results has implications for industrial adoption. The ability to deploy STMo monitors seamlessly from models to FPGA and ASIC platforms demonstrates the general suitability of our approach for safety- and security-critical platforms as in automotive and robotics systems. While the additional hardware-costs currently prevent an industrial exploitation the benefits of the hardware-based solution in terms of configurability, adaptability and security might soon overcome the disadvantage of the higher costs.

I.4 Fortschritt bei anderen Stellen

Runtime monitoring for real-time systems is a broad and still a hot topic. In the recent time it also achieved further advancements by other researchers. Some of the recent developments in this area are listed below.

1. Singh, N., Pal, S., Leupers, R., Merchant, F., & Rebeiro, C. (2024). PROMISE: A Programmable Hardware Monitor for Secure Execution in Zero Trust Networks. *IEEE Embedded Systems Letters*, 16(4), 433–436. <https://doi.org/10.1109/LES.2024.3354831>

The work proposes a programmable hardware runtime monitor for secure execution in Zero Trust networks. While the framework does not emphasize formal specification properties, it introduces a reconfigurable co-processor that monitors microarchitectural signals to generate device trust scores. In contrast, the STMo approach uses timing annotations and formal timing specifications (STMoSL) for monitoring the timing behavior rather than the micro-architectural signals and security scoring.

2. Yu Wang, Jinting Wu, Haodong Zheng, Zhenyu Ning, Boyuan He, and Fengwei Zhang. 2023. Raft: Hardware-assisted Dynamic Information Flow Tracking for Runtime Protection on RISC-V. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23)**, Hong Kong, Hong Kong—October 16–18, 2023. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3607199.3607246>

This work illustrates a different direction for hardware-assisted runtime monitoring on RISC-V. Raft employs a dedicated coprocessor attached to the Rocket core to perform Dynamic Information Flow Tracking (DIFT), with compiler-assisted instrumentation to propagate taints and detect security violations at runtime. This architecture targets data- and control-flow attack prevention, and achieves low overhead by offloading checks to the coprocessor. In contrast, the Statistic-Timing Monitor (STMo) approach developed in this work focuses on timing integrity using specification-driven synthesis of parameterizable RTL modules directly integrated into the FPGA or SoC fabric.

- Baumeister, J., Finkbeiner, B., & Scheerer, F. (2025). Active Monitoring with RTLola: A Specification-Guided Scheduling Approach. arXiv. <https://arxiv.org/abs/2507.20615>

Baumeister et al. (2025) introduce Active Monitoring with RTLola, which extends stream-based monitoring by adding specification-guided scheduling. Instead of passively sampling all data at fixed rates, the monitor actively prioritizes sensor inputs based on the specification and its current state, enabling faster detection of violations. Both the STMo approach and the RTLola approach highlight the importance of specification languages for reliable runtime monitoring but address different dimensions: RTLola targets the adaptive scheduling, whereas STMo targets the measured timing behavior monitoring through the statistics approach.

I.5 Literaturverzeichnis

- [1] T. Weilkiens, SYSMOD - The Systems Modeling Toolbox, Tim Weilkiens, 2020.
- [2] CBMC, „Bounded Model Checking for Software,“ 2025. [Online]. Available: <http://www.cprover.org/cbmc/>.
- [3] J. R. Hauser, „berkeley-softfloat-3,“ 2023. [Online]. Available: <https://github.com/ucbar/berkeley-softfloat-3>.
- [4] opencores, „Fixed-point Quadratic Polynomial,“ 2023. [Online]. Available: <https://opencores.org/projects/quadratic>.
- [5] opencores, „Generic FIR Filter,“ 2023. [Online]. Available: https://opencores.org/projects/quadratic_func.

I.6 Veröffentlichungen

1. N. R. Nanjundaswamy, G. Nitsche, F. Poppen and K. Grüttner, "RISC-V Timing-Instructions for Open Time-Triggered Architectures," 2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), Porto, Portugal, 2023, pp. 211-214, doi: 10.1109/DSN-W58399.2023.00058. keywords: {Schedules;Program processors;Embedded systems;Computer architecture;Software;Real-time systems;Timing;Time-triggered Architecture;RISC-V ISAX;Temporal Behaviour ;Run-Time Monitoring}
2. Trust is Good, Monitoring is Better: FPGA- & TEE-Based Monitoring for Malware Detection Friederike Bruns, Georg Gläser, Florian Kögler, Jonas Lienke, Nithin Nanjundaswamy, Gregor Nitsche, Behnam Perjikolaei and Jörg Walter
3. Nitsche, Gregor and Thirunavukkarasu, Arunachalam and Ravani Nanjundaswamy, Nithin and Schmedes, Rolf and Grüttner, Kim (2023) Execution Path Timing Statistics-based Safeguarding & Online Monitoring for Embedded Software Anomaly Detection. Tag der vertrauenswürdigen Elektronik 2023, 2023-05-10, Hannover, Deutschland.

4. Nitsche, Gregor and Thirunavukkarasu, Arunachalam and Ravani Nanjundaswamy, Nithin and Schmedes, Rolf and Grüttner, Kim (2023) Execution Path Timing Statistics-based Safeguarding & Online Monitoring for Embedded Software Anomaly Detection. edaWorkshop23, 2023-05-08 - 2023-05-09, Hannover, Deutschland.