Non-intrusive Runtime Verification within a System-on-Chip*

José Rufino, António Casimiro
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal; email: jmrufino@ciencias.ulisboa.pt

Felix Dino Lange, Martin Leucker, Torben Scheffel, Malte Schmitz, Daniel Thoma
Institute for Software Engineering and Programming Languages, Universität zu Lübeck, Lübeck, Germany; email: lange,leucker,scheffel,schmitz,thoma@isp.uni-luebeck.de

Abstract

This paper describes how to enrich a System-on-Chip (SoC) design by flexible monitoring capabilities allowing to analyze the system's execution for ensuring safety requirements. To this end, a general SoC architecture is described enriched by observation means. Moreover, it is described how verification properties expressed in a temporal stream-based specification language can be translated into a monitor expressed in a hardware description language (Verilog) checking the underlying property. Finally, the link between the SoC and the monitoring unit is explained. Overall, a self-observing system is obtained that works coherently with the SoC.

Keywords: SoC runtime verification

1 Introduction and Motivation

Autonomous vehicles are paving their way into application domains as diverse as: terrestrial, aerospace, maritime and submarine. They include a System-on-Chip (SoC), hosting an on-board computing system, to control the vehicle and ensure the fulfillment of its mission.

In general, those control functions are extremely complex, with strict real-time requirements. Interaction with the environment and operation in harsh or uncertain contexts are potential sources for lack of determinism. In any case, the correctness of the overall system is paramount, and safety should be ensured at all times.

Runtime Verification (RV) [1, 2] assumes herein great relevance, since it adds an extra layer of protection, assessing the system against a previously defined specification, checking whether timing and safety properties are satisfied or violated. Most of the current RV techniques require the modification of the application source code. Although software-based instrumentation is reasonable for larger systems, the requirements that characterise these vehicular systems may pose an unsurpassed challenge for runtime verification in such kind of systems. Other techniques, such as system and/or function call interception, are also not free from intrusiveness.

In this context, the concept of Hardware-based Observability, a non-intrusive observation and runtime verification technique, assume particular relevance. More precisely, the underlying idea is that a safety-critical system should be enriched by observation and analysis/monitoring techniques directly on the core system itself. Thus, they should become a part of the SoC. The direct combination allows perfect observability of the functional system. The allocation of hardware resource for the analysis further ensures that the monitoring does not affect the execution of the functional system.

While SoC are traditionally specified in hardware description languages like VHDL [3] or Verilog [4], the specification of verification properties should ideally be performed in some high-level domain specific language. Recently, the authors hosted at Lübeck introduced the temporal stream-based specification language TeSSLa [5] which is especially designed for specifying correct program executions. In this paper, we describe how TeSSLa specification can be translated into a hardware description language and integrated into a SoC for performing basic verification tasks. Overall, we obtain a self-observing system that works coherently with the SoC.

The paper is organized as follows. Section 2 presents hardware-based observability monitors. Section 3 focuses on an introduction to TeSSLa while Section 4 discusses its translation into a Verilog format. Section 5 evaluates the work done. Section 6 describes the related work and Section 7 presents some concluding remarks and future research directions.

2 Non-Intrusive Observation and Runtime Verification

The classical approach to runtime verification implies the instrumentation of the functional system software components: small pieces of software, acting as observers, are added to

*This work was partially supported by FCT, through funding of LASIGE Research Unit, ref. UID/CEC/00408/2013, and by FCT/CAMPUS FRANCE (PHC PESSOA programme), through the transnational cooperation project 3732 (PT) / 37932TF (FR), Non-intrusive Observation and RunTime verification of cyber-pHysical systems (NORTH). This work integrates the activities of COST Action IC1402 - Runtime Verification beyond Monitoring (ARVI), supported by COST (European Cooperation in Science and Technology). This work was partially supported by the BMBF project ARAMIS II with funding ID 01 IS 16025 and EU H2020, through project 732016, COEMS (Continuous Observation of Embedded Multicore Elements).

assess their state in runtime. Software-based instrumentation inherently disturbs the system, namely with respect to timing properties, which are crucial to system design.

2.1 Hardware-based Observability

The demand for non-intrusive observability justifies, per se, the interest in hardware-based methods, powered by: the usage of reconfigurable logic, supported on FPGA special purpose observers [6, 7, 8]; the raw availability of integrated observation resources [9, 10]. By nature, hardware-based system observation is completely non-intrusive and can be made, by design, extremely effective.

The architecture described in Figure 1 describes the functional system platform, implemented as a SoC architecture and how runtime observation and monitoring features can be integrated non-intrusively, meaning execution of runtime verification actions does not disturb the execution of the functional system software components. Probing the processor-cache interfaces should allow an higher accuracy in the observation of software components execution.

2.2 Observer Entity

The Observer Entity defined by the architecture of Figure 2 aims to support the non-intrusive observation and runtime verification of an associated functional system, therefore enabling the verification in runtime that its properties are being fulfilled and that no design assumption is being violated.

The Observer Entity is plugged to the platform where the functional system software components execute, and comprises the hardware modules of Figure 2: Bus Interfaces, capturing all physical bus activity, such as bus transfers or interrupt vectors; Management Interface, enabling observer entity configuration; Configuration, storing the dynamic set of events; the System Observer itself, detecting events of interest; Monitor, which detects possible violations of the specified system behaviour; Time Base, which allows to time stamp the events of interest.

2.3 System Observing Mechanisms

The System Observer collects, in runtime, from the functional system bus interfaces, all the addressing/data information to detect events of interest set by configuration, performed statically (offline) or dynamically, while the system is executing.

When an event of interest (e.g., the fetch of a specific instruction or a read/write access to a given variable in the memory) is detected, it is timestamped with the instant of occurrence, as obtained from the Time Base module, and supplied to every downstream block awaiting for that event. A unique identifier (obsID) is assigned to each observed event, being an event composed by the tuple:

$$evt_{obsID} =< a_{obs}, v_{obs}, t_{obs} >$$

where: a_{obs} is the address observed from the functional system bus interface that matches a given event specification; v_{obs}, the corresponding observed value (e.g., instruction coding or data value); t_{obs}, is the attached timestamp.

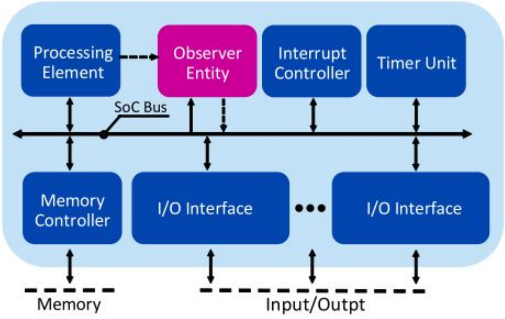

Figure 1: Generic SoC architecture and Observer Entity

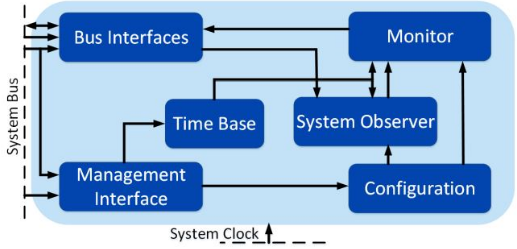

Figure 2: Observer Entity architecture

2.4 Monitoring Mechanisms

A divide and conquer strategy is used in the definition and design of a minimal set of hardware-based essential blocks for the synthesis of runtime verification mechanisms. A set of basic monitors, encompassing essential runtime verification actions, in both value and time domains, is detailed in [11]. These monitors can be instantiated as required. Additional blocks (selectors, transformers and past-time event registers) complement and enlarge the functionality provided by the basic monitors. The right combination of these building blocks should be able to provide the necessary and sufficient mechanisms for the runtime verification of any functional system.

3 An introduction to TeSSLa

TeSSLa [5] is a temporal stream-based specification language that is designed for monitoring real-time signals and has already been used to build monitors for Runtime Verification [12]. TeSSLa reasons over asynchronous input streams and provides a rich data domain (Boolean, integer, real). Monitors specified in TeSSLa can observe events, that were emitted with different speeds and with different delays. TeSSLa supports signals and event streams. An event stream is only allowed to be defined for a finite number of timestamps in a finite interval, while a signal stream defines a value for every point in time.

The basic concept of TeSSLa is deriving internal or output streams by applying functions to already existing streams. A stream can be defined declaratively as can be seen in the following example of a TeSSLa specification:

```
def maximum := max(x1, x2)
def max(a,b) := if a > b then a else b
```

The specification contains two input streams $x1$ and $x2$ and creates a new stream $maximum$ which always contains the larger value of $x1$ and $x2$. Note that it is possible to define macros (i.e. $max(a, b)$) that can be used to define more sophisticated properties.

A complete list of all functions can be found in [5]. For example can basic arithmetic function (like the comparison of two numbers) be lifted. The lifted function is able to reason over streams instead of i.e. integers. For timing properties it is possible to generate a stream of timestamps corresponding to the current value of another stream:

```
def timeOfx := time(x)
```

TeSSLa is useful for the approach of hardware-based monitoring because it is especially designed for monitoring streams and can be directly translated into hardware descriptions as is explained in the following.

4 Translation into Verilog

Figure 3 shows the approach of Non-intrusive Runtime Verification within a System-on-Chip. The TeSSLa compiler translates the TeSSLa specification into a dependency graph. The dependency graph contains the necessary information to generate Verilog code, which is used to synthesize the monitor in FPGA hardware. There are five different operators that have to be considered for the translation. Every operator can directly be implemented as a node in the dependency graph of a TeSSLa specification and the nodes can be connected via message parsing. To show that this direct translation is generally possible, two cases have to be considered.

Without recursion: If there are no recursions in a TeSSLa specification, its dependency graph is known to be a directed, acyclic graph [12]. To make sure that there is always at least one node that is able to write, extra events, called progress events, are introduced. Their purpose is to inform nodes downstream about the absence of events. From that follows a constant event throughput at all times. A formal proof can be found in [13].

With recursion: There are two operators in TeSSLa that have recursive behaviour. The $last()$ operator returns a stream with the last value of another stream based on a trigger signal. The $delay()$ operator delays a stream by a given amount of time and can be reset by a signal stream. Because both the trigger signal stream and the reset signal stream cannot be recursive, it is guaranteed that a progress exists at all times.

This shows that every TeSSLa specification produces the same output independently of timed reordering and it is therefore possible to translate into evaluation engines implemented in Verilog.

5 Use Case Integration

A use case in the domain of aerospace is the observation of a navigation system of a satellite. The execution time of different tasks with different priorities has to be observed, because sometimes the execution takes longer than expected. If a task exceeds the expected execution time, it has to be canceled so other task can be executed in time. However, if the same task is failing three times in a row, this is considered an error, because the calculation of the trajectory of the satellite needs the result of this task at least every third execution.

In order to monitor this behavior, we need to check the runtime of the task, compare the timestamps and count the number of failed task executions. The runtime of the task can be gathered by instrumenting the hardware of the Leon processor as described in Section 2 and can be passed as streams of events to a monitor on the FPGA. The two streams contain the events of starting ($call$) and finishing ($return$) the task. The runtime of the tasks can be calculated and compared with TeSSLa:

```
def runtime := on(return,
   time(return) - time(call))
def count_violations :=
   if runtime > threshold
   then resetcount(runtime, false)
   else
   resetcount(runtime, true)
```

Note that the macro $on(x, y)$ assures that the stream $runtime$ is only updated, if a new $return$ event was sent. $resetcount(trigger, reset)$ returns the counted number and is reset every time the second argument is $true$. The full code for the macros is not shown in this paper due to paper size limitations. An error is declared, if the threshold is violated three times in a row:

```
def error :=
  if count_violations > 3 then true
  else false
```

This specification is then translated into a dependency graph by the TeSSLa compiler. The dependency graph can be used to generate a hardware specification as described in section 4. With this setup it is possible to observe the activity of the satellite for an unlimited amount of time and gather information about the runtime violations of certain tasks.

6 Related Work

The application of non-intrusive runtime monitoring to embedded systems has been discussed in [6,14] and, more specifically, in safety critical environments [15]. Configurable minimally intrusive event-based frameworks for dynamic runtime monitoring have been developed [16]. Additionally, the RV concept has been applied to autonomous systems [17] and to diagnose multi-processor SoC [18]. However, to the extent of our knowledge, no previous work has exploited how a TeSSLa specification can be translated into a hardware description language and integrated into a SoC.

7 Conclusion

We propose an approach on how to combine hardware-based non-intrusive observation of a System on Chip with the high-level temporal stream-based specification language TeSSLa. With its easy to read C-style TeSSLa can be used to describe properties much more intuitively than directly in hardware

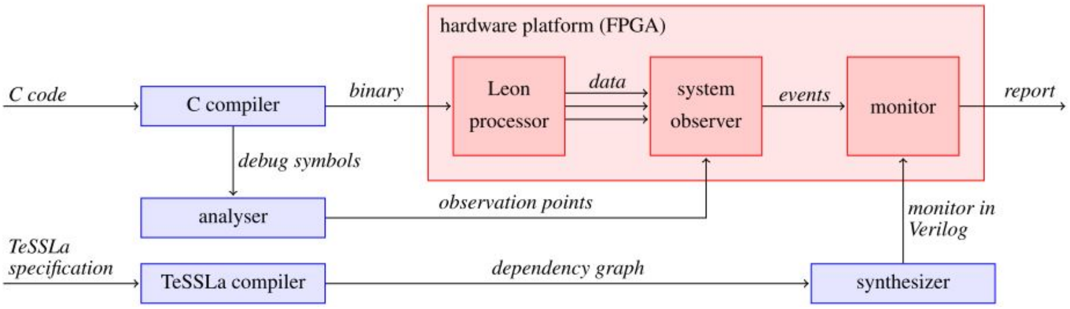

Figure 3: An overview over our approach

description languages. It can be shown that TeSSLa specifications are directly translatable into a hardware description language like Verilog.

Hardware-based observation is especially useful in domains with long observation times. Therefore we introduce the use case of task runtime observation of a satellite navigation system to show a possible application of this approach.

This paper is the first step towards integrating TeSSLa into a SoC. The use case prototype, showing the feasibility of hardware-based observation within a SoC, needs further work, namely with regard to: the exploitation of the monitoring infrastructure [11]; the translation from TeSSLa to a hardware description language; the definition of an effective algorithm for the direct translation from the TeSSLa specification.

References

[1] M. Leucker and C. Schallhart (2009), *A brief account of runtime verification*, The Journal of Logic and Algebric Programming, vol. 78, pp. 293–303.

[2] Y. Falcone, K. Havelund, and G. Reger (2013), *A Tutorial on Runtime VerificationEngineering*, in Dependable Software Systems, vol. 34, pp. 141–175. Marktoberdorf, Germany: IOS Press Ebooks.

[3] IEEE (2018), *1076.1-2017 - IEEE Standard VHDL Analog and Mixed-Signal Extensions*.

[4] IEEE (2018), *1800-2017 - IEEE Standard for SystemVerilog– Unified Hardware Design, Specification, and Verification Language*.

[5] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma (2018), *TeSSLa: a temporal stream-based specification language*, in International Colloquium on Theoretical Aspects of Computing (ICTAC). Submitted for publication.

[6] C. Watterson and D. Heffernan (2007), *Runtime verification and monitoring of embedded systems*, Software, IET, vol. 1, Oct. 2007.

[7] J. C. Lee, A. S. Gardner, and R. Lysecky (2011), *Hardware observability framework for minimally intrusive online monitoring of embedded systems*, in Proc. 18th Int. Conf. on Engineering of Computer Based Systems, (Las Vegas, USA), pp. 52–60, IEEE.

[8] R. C. Pinto and J. Rufino (2014), *Towards non-invasive runtime verification of real-time systems*, in 26th Euromicro Conf. on Real-Time Systems - WIP Session, (Madrid, Spain), pp. 25–28.

[9] ARM (2013), *ARM CoreSight Architecture Specification*, Cambridge, England, 2.0 ed.

[10] R. Backasch, C. Hochberger, A. Weiss, M. Leucker, and R. Lasslop (2013), *Runtime verification for multicore SoC with high-quality trace data*, ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 18, p. 18.

[11] J. Rufino (2018), *Runtime verification monitors*, tech. rep., Faculdade de Ciências da Universidade de Lisboa, Portugal.

[12] N. Decker, P. Gottschling, C. Hochberger, M. Leucker, T. Scheffel, M. Schmitz, and A. Weiss (2017), *Rapidly adjustable non-intrusive online monitoring for multicore systems*, Brazilian Symposium on Formal Methods, pp. 179–196, Springer.

[13] M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, and A. Schram (2018), *TeSSLa: Runtime verification of nonsynchronized real-time streams*, in ACM Symp. on Applied Computing (SAC), (Pau, France), ACM.

[14] T. Reinbacher, M. Fugger, and J. Brauer (2014), *Runtime verification of embedded real-time systems*, Formal Methods in System Design, vol. 24, no. 3, pp. 203–239.

[15] A. Kane, O. Chowdhury, A. Datta, and P. Koopman (2015), *A case study on runtime monitoring of an autonomous research vehicle (ARV) system*, in Proc. 15th Int. Conf. on Runtime Verification, (Vienna, Austria).

[16] J. C. Lee and R. Lysecky (2015), *System-level observation framework for non-intrusive runtime monitoring of embedded systems*, ACM Transactions on Design Automation of Electronic Systems, vol. 20, no. 42.

[17] G. Callow, G. Watson, and R. Kalawsky (2010), *System modelling for run-time verification and validation of autonomous systems*, in Proc. 5th Int. Conf. on System of Systems Engineering, (Loughborough, UK).

[18] P. Wagner, T. Wild, and A. Herkersdorf (2017), *DiaSys: Improving SoC insight through on-chip diagnosis*, Journal of Systems Architecture, vol. 75.