Hardware runtime verification of embedded software in SoPC

Dimitry Solet, Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, Sébastien Pillement

Hardware Runtime Verification of Embedded Software in SoPC

Dimitry Solet*, Jean-Luc Béchennec†, Mikaël Briday†, Sébastien Faucou†, Sébastien Pillement*
*UMR 6164 IETR, Université de Nantes
†UMR 6597 IRCCyN, Université de Nantes & CNRS

***Abstract*—This paper discusses an implementation of runtime verification for embedded software running on a System-on-Programmable-Chip (SoPC) composed of a micro-controller and a FPGA. The goal is to verify at runtime that the execution of the software on the micro-controller conforms to a set of properties. To do so, a minimal instrumentation of the software is used to send events to a set of monitors implemented in the FPGA. These monitors are synthesised from a formal specification of the expected behavior of the system expressed as a set of past-time linear temporal logic (ptLTL) formulas.**

I. Introduction

Traditional techniques used to improve the correctness of embedded software include program verification (proof, model checking) and testing. Program verification offers a complete coverage of the system at early design stages, so it is usually not able to detect defects introduced during later stages (compilation, link, etc.). Testing operates on the "real" system but does not offer a complete coverage of the possible behaviours of the system. Thus, none of them offers a full coverage of the system lifecycle. In particular, the operational phase is not covered by these techniques.

For a number of embedded systems such as totally autonomous systems or safety-critical systems, it is important to have the capacity to detect and react to faults occurring at runtime. To achieve this goal, these systems can use well-known fault-tolerant design techniques such as runtime monitoring for fault detection or triple modular redundancy for recovery. The main drawback of these techniques is their cost, both in terms of consumption of runtime resources (space, energy, computation time, memory) and engineering.

In this paper, we describe a runtime monitoring technique to detect fault occurring at runtime in System-on-Programmable-Chip (SoPC). A SoPC integrates one, or more, hardcore processor combined with a programmable logic circuit. Examples of SoPC on the market include Xilinx Zynq [3] or Microsemi Smartfusion [1]. In these architectures, the programmable logic circuit is rather used to implement hardware accelerators. Our idea is to use this circuit to implement safety devices. This approach should allow to limit the consumption of runtime resources. In this paper, we focus on a first type of safety devices: runtime monitors. To limit the cost in term of engineering, we rely on a fully automatised chain to synthesise these monitors, using the algorithms and tools developed in the framework of the runtime verification theory.

The paper is organised as follow : in Section 2, we give an overview of runtime verification, runtime monitoring, and we presents some related works; in Section 3, we provide some background on the past-time linear temporal logic (ptLTL) used to formally specify the monitored properties; in Section 4 we discuss the design and implementation of the runtime monitoring framework; in Section 5, we present a small case study. Lastly, we highlight some future works and conclude the paper in Section 6.

II. Overview

A. Runtime Verification and Runtime Monitoring

Runtime Verification (RV) is usually classified as a lightweight formal methods dedicated to the synthesis of runtime monitors from specifications [10]. Specifications are usually given in the form of temporal logic formulas and transformed step by step into a monitor. This monitor can be used to analyse the execution trace of a system and to decide if this trace conforms or not to the specification. The analysis can be performed either at runtime or offline. In this paper, we are interested in runtime analysis. We use RV techniques to synthesise monitors. Then we plug these monitors in a Runtime Monitoring framework.

As illustrated in Figure 1, a Runtime Monitoring framework is composed of three main components:

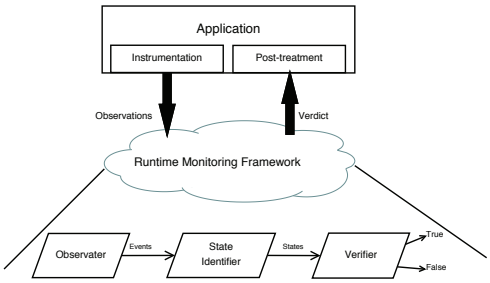

Fig. 1. Architecture of a runtime monitoring framework

- The observer extracts relevant information from the observation of the execution of the monitored system in the form of a stream of events;
- The state identifier is used to build the state of the system from the event stream. The sequence of states is the execution trace;
- The checker computes a verdict from the input trace.

All these components can be implemented either as software, hardware, or a mix of both.

A software implementation is straightforward but requires the instrumentation of the system by adding relevant code. This instrumentation has a cost, especially in term of execution time and memory footprint [8]. Thus, in an embedded system with severe resource constraints, a software implementation might not be affordable. Moreover, a software implementation presents the disadvantage of sharing common failure case with the monitored system.

A hardware implementation consists in plugging a dedicated hardware device. This device is able to observe at least read-/write operations on the system bus between the cores, memory and peripherals. It can then relay these signals to the monitors, also implemented as circuits. This allows to circumvent the limitations listed above by approaching a zero overhead on software execution and by enforcing spacial isolation between the application and the monitors. However, it requires to use a fully dedicated hardware platform (ASIC or FPGA). Even in this case, it is difficult to achieve a complete observability of the system without using an emulator [5]. Lastly, the properties used to design the monitors must be expressed in terms of low level signals, which can be cumbersome for complex software.

In this paper, we explore a hybrid approach based on a minimal instrumentation of the application software used to send signals to a hardware monitoring framework.

B. Related works

P2V [11], [12] is a compiler that translates a specification expressed in PSL (Property Specification Language) into monitors written in Verilog code. P2V performs the verification of a software program, written in C language, that is executed on a processor which is synthesised into a FPGA (soft-core). To generate a monitor from a PSL property, P2V uses information generated by the C compiler to understand the mapping between C and machine code. This allows not to instrument the software. Therefore, P2V provides a runtime verification solution with zero-overhead in software. However, in term of computation power and energy consumption, soft-core are not as efficient as hardcore. Then, in an industrial context, soft-core are rarely used.

A second work [15] presents a runtime verification approach for micro-controller binary code. This approach uses the ptLTL logic to express the specifications to verify. The target system and monitors are synthesised into a FPGA. This allows to observe passively the system bus in order to compute a verdict about a specification. This work is focused on the runtime verification of micro-controller code. The target system is a soft-core. Thus, this method presents the same features as the previous work.

In [13], authors presents the BusMOP framework which generates monitors for the runtime monitoring of bus activities. Monitors are synthesised into a FPGA which is plugged on a PCI (Peripheral Component Interconnect) bus in order to check communications between each peripheral. Properties to check are expressed in extended regular expression (ERE) and ptLTL. The aim of this framework is to monitor COTS (Commercial-Off-The-Shel) peripheral by spying passively the PCI bus. Thus, their approach has zero CPU runtime overhead.

More recently, [5] introduces an approach for the runtime verification of multicore SoCs. An external emulator of the system is used. This emulator, called *hidICE*, replicates the SoC bus master cores. It is synchronised with the SoC to observe data which can change the program flow: peripheral data, clock, interrupts and bus wait states. From the emulation, program counter and instructions of each core are rebuilt. These information is sent to a trace analyser which generates a trace composed of propositions. Monitors take these propositions to evaluate the given specifications. Monitors are designed from a Linear-time Temporal Logic (LTL) property. This approach has been evaluated on FPGAs with soft-cores, however the authors expect that their approach will be supported on SoPC with hard-cores. We can note that the aim of their approach is to provide a runtime verification framework to use during the development phase and it cannot to be used for the operational phase of the system.

C. Contributions

In this paper, we provide the following contributions: first, we discuss on the design and implementation of the architecture of a runtime monitoring mechanism for SoPC platforms. To the best of our knowledge, this is the first work exploring RV for SoPC. Some implementation choices might be specific to our target architecture but we believe that the discussion is general enough to be useful in a broader context. Second, we describe a chain that, given a specification in the form of a temporal logic formula, generates monitors in the form of HDL programs that can be hosted in our runtime monitoring framework. Third, we show the feasibility of our approach on a small case study.

III. Background

A. Choice of a specification language

As shown in the section II-B, several specification languages can be used to express properties that can be used to generate runtime monitors, including LTL, PSL and ptLTL. LTL is an infinite trace linear future time temporal logic [14]. PSL is a specification language for hardware design that also includes an infinite trace linear time temporal logic with both future and past time modalities [4]. ptLTL is a finite trace linear past time temporal logic [9]. In this paper we choose to use ptLTL.

Comparing expressiveness, PSL is strictly more expressive than LTL which is in turn strictly more expressive than ptLTL. However, in the context of runtime verification, we do not need all the expressive power of LTL or PSL because we deal only with finite traces. There exists LTL or PSL formulas in which the satisfaction can not be deduced on a finite trace. These formulas are not amenable to runtime verification. On the other hand, all ptLTL formulas are amenable to runtime verification.

Moreover, as noted in [9], ptLTL semantics can be defined recursively in a way that the satisfaction of a formula and a trace can be computed along the trace looking only one step backward. From this definition, as explained in section IV-C, it is straightforward to deduce an efficient hardware implementation of a satisfaction algorithm for ptLTL.

Further arguments on the interest of using ptLTL in the context of runtime verification can be found in [16].

B. Syntax and semantics of ptLTL

ptLTL extends propositional logic with the following temporal past-time operators:

$\odot F$:	previously F
$\boxdot F$:	always F in the past
$\diamondsuit F$:	eventually F in the past
$F_1 S_s F_2$:	F_1 strong since F_2
$F_1 S_w F_2$:	F_1 weak since F_2

These ptLTL operators are similar to LTL operators : previously is analogous to next, always in the past is analogous to always, eventually in the past is analogous to eventually and strong/weak since is analogous to strong/weak until.

In addition of these operators, it is suitable to introduce monitoring operators which are used to help the requirement specification writing. These operators do not add expressiveness to ptLTL, in fact, these operators can be expressed by the use of the standard ptLTL operators:

start F :	$\uparrow F = F \wedge \neg \odot F$
end F :	$\downarrow F = \neg F \wedge \odot F$
strongly in $[F_1 F_2]$:	$[F_1; F_2)_s = \neg F_2 \wedge ((\odot \neg F_2) S_s F_1)$
weakly in $[F_1 F_2]$:	$[F_1; F_2)_s = \neg F_2 \wedge ((\odot \neg F_2) S_w F_1)$

The semantics of a ptLTL formula is defined over a trace associated with an execution of a system. Let AP be a set of atomic propositions. Let λ be a labelling function mapping a state of the system to a subset of AP. Let s be a state of the system. We note $\sigma = \lambda(s)$. When an event occurs in the system, it triggers transition from state s_i to state s_{i+1}. A trace of the system after n events is the finite sequence $t = \sigma_1 \sigma_2 ... \sigma_n$. We let t_i denote the prefix $t_i = \sigma_1 \sigma_2 ... \sigma_i$ of t for $1 \leq i \leq n$.

Let p be a proposition in AP, F, F_1, F_2 be ptLTL formulas and $t = \sigma_1 \sigma_2 ... \sigma_n$ a trace in 2^{AP*}. The semantics of a ptLTL formula over t is defined as follows:

$t \vDash p$	iff $p \in \sigma_n$,
$t \vDash \neg F$	iff $t \nvDash F$,
$t \vDash F_1 \wedge F_2$	iff $t \vDash F_1$ and $t \vDash F_2$,
$t \vDash \odot F$	iff $n > 1$ and $t_{n-1} \vDash F$,
$t \vDash \diamondsuit F$	iff $t \vDash F$ or ($n > 1$ and $t_{n-1} \vDash \diamondsuit F$) ,
$t \vDash \boxdot F$	iff $t \vDash F$ and ($n > 1$ implies $t_{n-1} \vDash \boxdot F$) ,
$t \vDash F_1 S_s F_2$	iff $t \vDash F_2$ or ($n > 1$ and $t \vDash F_1$ and $t_{n-1} \vDash F_1 S_s F_2$) ,
$t \vDash F_1 S_w F_2$	iff $t \vDash F_2$ or ($t \vDash F_1$ and ($n > 1$ implies $t_{n-1} \vDash F_1 S_s F_2$).

C. An example of ptLTL specification

In [2], patterns about property specification for finite-state verification are given. Let us take as example one of this pattern and express it in a ptLTL formula.

We consider the following pattern where P, Q and R are atomic propositions: "Always P is false after Q until R". This is illustrated on the Figure 2:

The ptLTL formula which expresses this pattern is:

$$F = \boxdot([Q, R)_s \rightarrow \neg P)$$

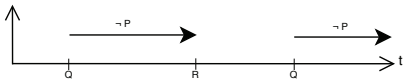

Fig. 2. Diagram of the example

IV. Implementation

In this section, we will focus on the implementation of a runtime monitoring framework into a system based on a SoPC, then a system which integrates a micro-controller and a FPGA fabric. The application runs on the micro-controller and the monitor is implemented into the FPGA. As illustrated in Figure 3 the micro-controller and the FPGA communicate through interrupt signals and a communication bus.

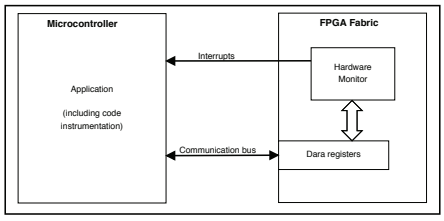

Fig. 3. Architecture of the SoPC hardware platform

The communication bus between the micro-controller and the FPGA allows the micro-controller to access data, control and status registers in the FPGA. Data registers can be used to receive information from the application. The interrupt controller on the other way allows to send a verdict from a monitor to the micro-controller. The interrupt handler has the responsibility to manage the outcome of a verdict.

Figure 4 gives an overview of the framework. Events are generated by the application and transmitted to the FPGA in order to be stored in the register file. Then, the State Identifier analyses the events in order to compute a trace. Finally, the ptLTL checker verifies that the trace respects the ptLTL property and sends back the related verdict to the system.

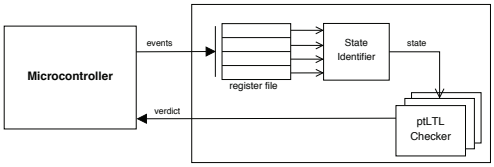

Fig. 4. Overview of the runtime monitoring framework

This runtime monitoring framework raises some questions :

- How to observe the system with the application instrumentation?
- How to identify the states?
- How to generate the ptLTL checker from a property?
- How to synchronize the monitor and the application?

A. Observation of the application

Our framework provides two mechanisms to enable the observation of the application by the verification module. Both relies on a set of data registers implemented in the FPGA.

The first mechanism consists in mapping application variables to FPGA registers. The main advantage is that we do not need to add any instruction to the code of the application. The main drawback is that any access to such a variable (including read access) requires a data transfer through the communication bus between the microprocessor and the FPGA. These transfers are usually slower than transfer to SRAM banks.

The second mechanism consists in adding dedicated instruction in the code of the application to write data to FPGA registers. The main advantage is that we pay the price of a data transfer to the FPGA only when needed. Moreover, for fixed size data such as boolean values, it allows to minimize the usage of registers by mapping several data to a single register.

We have to study thoroughly both approaches in order to provide the designer with relevant informations when facing the choice to use one of these approaches.

B. State identification

The state identification is performed by the analysis of the registers which are written by the monitored application. This analysis is used to update the value of each atomic proposition.

The design of state identifiers relies on the formalisation of the atomic propositions which can be specified as:

- The comparison between the values of two registers or between the value of a register and a constant.
- The occurrence of a software event.
- The access of a specific state that is given by the use of a FSM (Finite-State Machine).

1) Comparator: It performs a comparison between two data. A data is either the value of a monitored variable or a constant. At least one of these data is a monitored variable. This design is similar to the comparator design of [15].

2) Event detector: The event detector is a sequential system to detect the occurrence of a software event. The event to monitor is a bit of the software event register (noted $Event$). A D flip-flop delays the event of 1 cycle ($Event^{-1}$) and a comparator between $Event$ and $Event^{-1}$ gives the result of the occurrence of the event (Figure 5).

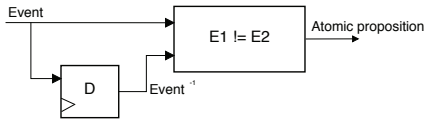

Fig. 5. Design of an event detector

3) Finite-state machines: A FSM can be used to compute more complex atomic propositions. In this case the value of an atomic proposition is defined according to the current state of the FSM. The triggering conditions for each transition are composed of the occurrence of software events. We can note that the use of FSM allows to define complex atomic propositions which leads to simplify the specification.

C. ptLTL Checker

The ptLTL checker is the hardware structure which allows to compute a ptLTL formula. It is based on the satisfaction algorithm which is deduced directly from the ptLTL semantics [9]. The satisfaction of a ptLTL formula is realised by the computation of each sub-formulas.

The memory used for the satisfaction of a ptLTL formula is proportional to the number of sub-formulas, and the computation time is proportional to the depth of the ptLTL formula.

The recursive semantics gives directly the HDL code of each ptLTL operation. Let us take the example $F := [a; b)_s \rightarrow c$: "The observation of a implies that c is true until the observation of b". It is straightforward to identify the hardware design in Figure 6:

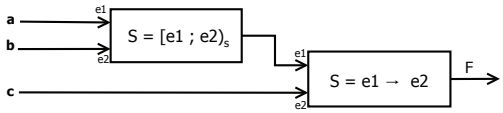

Fig. 6. Design of the ptLTL checker

Where c denotes the state of the operator output for one step backward.

The latency of the ptLTL checker is proportional to the depth of the ptLTL formula. To have a coherent verdict, the latency must to be lower than the clock period. Otherwise the solution is to split the ptLTL checker using D-flip-flop. In this way the verdict is coherent however the verification is performed in several clock period.

We assume that the depth of ptLTL formula from which the ptLTL checker is generated will respect this constraint.

D. Synchronisation

The verdict of the runtime monitoring is computed on the FPGA in parallel with the execution of the application. Without synchronisation, the application continues its execution and the verdict may be given too late. Thus, the architecture shall include a synchronisation mechanism. Two approaches are possible:

- *blocking approach*: When an event is set by the application, the application waits until the end of the verification. If the verdict is false, the interrupt request is received before to execute the next instruction. This method ensures the reliability of the execution but generates an overhead on the execution time of the software.
- *non blocking approach*: The application resumes its execution without waiting for the completion of the verification. If the verdict is false, the application can execute some instructions before the interrupt request.

The two approaches have advantages and are complementary. A trade-off between reliability and performance can be chosen by the designer. Our implementation supports both approaches. We have to study thoroughly this trade-off in order to provide the designer with relevant informations when facing this choice.

The blocking approach can be implemented either in *software* or in *hardware*.

The *software blocking approach* is the simplest to implement. It uses a status bit in the FPGA. When a data register of the FPGA is written, the status bit is unset. The application

polls the status bit while the verification is being performed, . Once the verdict is computed, the status bit is set and the application can resume.

The *hardware blocking approach* is possible if the communication protocol provides a synchronization signal to acknowledge a transfer. As long as this signal has not been set, the transaction is in progress and the bus master is in a wait state. Thus synchronisation can be achieved by delaying the signal after the completion of the computation of the verdict. The delay is dependent on the computation time of the verdict. As illustrated by Figure 7, this can be done with a timer. This is the solution used by our framework.

By setting the delay of the timer to 0, we obtain a *non blocking approach*.

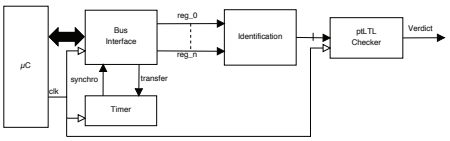

Fig. 7. Design of the synchronisation mechanism

E. Tool support

A Python program is currently under development. This program will enable to generate the VHDL code of the monitor from ptLTL formulas and description of atomic propositions.

The program will be a free software.

V. Case study

A. Presentation of the target platform: The SmartFusion 2 (SF2)

The SF2 [1] is a SoPC which integrates a 166 MHz ARM Cortex M3 processor and a flash-based FPGA fabric. The application code to verify runs on the processor and the FPGA fabric is used to implement the hardware monitors.

The communication between the processor and the FPGA is conformed to the AMBA bus specification. This open standard bus specification has been introduced by ARM Ltd. and gives rules to manage the communication between each peripherals in a SoC. The SF2 uses two different buses based on the AMBA specification:

- Advance High-performance Bus (AHB): it allows high performance pipelined operations and burst transfers (*i.e.* connection with an on-chip memory).
- Advance Peripheral Bus (APB): it is used with peripherals with low bandwidth that do not require pipelined operation nor burst.

The register interface in this case study is based on the APB specification.

The APB is a synchronous communication protocol based on the master/slave model where there is one master and several slaves peripherals. In our case, APB master is located in the MSS (Micro-controller SubSystem) and the APB slave is located in the FPGA fabric. An APB transfer is performed in at least two cycles.

This protocol provides a signal which allows to synchronise the running application with the verification hardware. When this signal is set to '1' the APB transfer is extended, then the running application is in a waiting state until the PREADY signal is set to '0'.

B. Overview of the example application

The software architecture of the proof-of-concept application to verify is shown in Figure 8. This small example is extracted from an industrial case-study of the automotive domain [7].

The application is composed of three tasks (T_0, T_1 and T_2). These tasks communicate through two buffers (b_0 and b_1). We define:

- s_{ij} the operation which indicates that the task T_i sends a message to the buffer b_j.
- r_{ij} the operation which indicates that the task T_i receives a message to the buffer b_j.

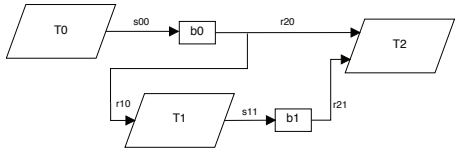

Fig. 8. Software architecture of the application

In this architecture, tasks T_0 produces data (coming from some other task not shown here). Task T_1 performs a computation based on the data produced by T_0. Task T_2 performs a coherency check on the output of T_1. To perfom this check, it has to know the input values of T_1. Thus, both inputs of T_2 needs to be correlated.

We say that buffer b_0 and b_1 are synchronised at time t when the value currently stored in b_1 has been computed by T_1 using the value currently stored in b_0. This property is associated with atomic proposition $synchro$.

We define two other atomic propositions:

- $Read_{start}$: true during a cycle, when T_2 read its first input;
- $Read_{end}$: true during a cycle, when T_2 read its second input;

We can now formally express the property that we want to monitor:

$F = \boxdot([Read_{start}; Read_{end})_s \rightarrow Synchro)$

C. Instrumentation of the application

We have to instrument the application and design the state identifier layer in order to identify the truth value of the three atomic propositions $Synchro$, $Read_{start}$ and $Read_{end}$.

For $Synchro$: the application is instrumented to send events s_{00}, r_{10} and s_{11} to the application. To process these events, we use the Moore machine described Figure 9. In the current version of our framework, this machine has to be hand-coded. As explained in [6], the instrumentation code can be added either at the application level or inside the kernel of the RTOS. The later solution is preferred because it allows to execute

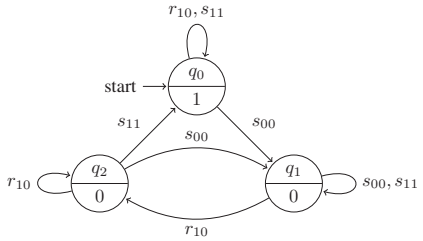

Fig. 9. Moore machine to compute the truth value of $Synchro$.

atomically the sending of the event and the corresponding action.

For $Read_{start}$: the application is instrumented to send an event when task $T2$ reads its first input. As explained above the atomic proposition $Read_{start}$ is true for a cycle whenever this event occurs. Thus to compute the truth value of $Read_{start}$ we use the event detector block described Figure 5. Let us underline that this design behaves correctly because the application can not generate two events in two consecutive cycles. This is impossible because sending an event through the APB bus takes two cycles.

For $Read_{end}$: we use the same design as for $Read_{start}$ with an event generated by the application when task $T2$ reads its second input.

The overall design is illustrated Figure 10.

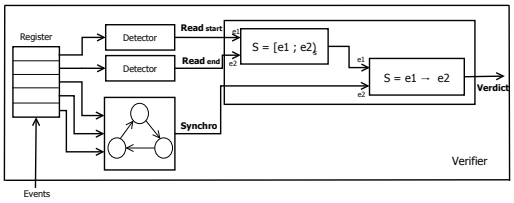

Fig. 10. Overall design of the verifier.

D. Implementation

We have implemented this verifier using our framework on the SF2 platform. The FPGA of the SF2 is composed of 56340 4-LUT (4 input Look-Up-Table) and DFF (D Flip-Flop). The verifier uses 71 4-LUT and 79 DFF. This is a very small footprint corresponding to less than 0.20% of the resources. Even in the case of a realistic application where the verifier will have to check more properties, the footprint should stay low. This should allow to use most of the resources of the FPGA to implement classical accelerators alongside the verifier.

VI. Conclusions and future works

This paper present a hardware implementation framework for runtime verification or embedded systems. We have explained the architecture of the framework and illustrated its use on a very simple example extracted from an industrial case-study. In the future we have to study realistic applications to gain a better understanding of the design choices offered by the framework. One of the target for this work is the kernel of Trampoline RTOS.

VII. Acknowledgment

This work is supported by the RFI "Électronique Professionnelle", Région Pays de la Loire - FRANCE within the framework of the SPARTES (Safety Programmable Application for Real Time Embedded System) project.

References

[1] Smartfusion2 soc fpgas. http://www.microsemi.com/products/fpga-soc/soc-fpga/smartfusion2. Accessed: 2016-02-25.
[2] Specification patterns. http://patterns.projects.cis.ksu.edu.
[3] Zynq-7000 all programmable soc. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. Accessed: 2016-02-25.
[4] Accellera. *Property Specification Language : Reference Manual*, 2004.
[5] Rico Backasch, Christian Hochberger, Alexander Weiss, Martin Leucker, and Richard Lasslop. Runtime verification for multicore soc with high-quality trace data. *ACM Trans. Des. Autom. Electron. Syst.*, 18(2):18:1–18:26, 2013.
[6] S. Cotard, S. Faucou, J.-L. Bchennec, A. Queudet, and Y. Trinquet. A data flow monitoring service based on runtime verification for autosar. In *IEEE Embedded Software and Systems (HPCC-ICESS)*, pages 1508–1515, 2012.
[7] Sylvain Cotard. *Contribution la robustesse des systmes temps réel embarqués multicœur automobile*. Thèse de doctorat, Université de Nantes, 2013. in french.
[8] Sylvain Cotard, Sébastien Faucou, and J.-L. Béchennec. A Dataflow Monitoring Service Based on Runtime Verification for AUTOSAR OS: Implementation and Performances. In *Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*, 2012.
[9] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 342–356. 2002.
[10] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.
[11] Hong Lu and A. Forin. The design and implementation of p2v, an architecture for zero-overhead online verification of software programs. (MSR-TR-2007-99):12, August 2007.
[12] Hong Lu and A. Forin. Automatic processor customization for zero-overhead online software verification. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 16(10):1346–1357, 2008.
[13] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time Systems Symposium (RTSS)*, pages 481–491, 2008.
[14] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS)*, pages 46–57, 1977.
[15] Thomas Reinbacher, Jörg Brauer, Martin Horauer, Andreas Steininger, and Stefan Kowalewski. Past time ltl runtime verification for microcontroller binary code. In *Formal Methods for Industrial Critical Systems (FMICS)*, pages 37–51. 2011.
[16] Grigore Rosu, Feng Chen, and Thomas Ball. Synthesizing monitors for safety properties: This time with calls and returns. In *Runtime Verification*, pages 51–68, 2008.