Enforcing Timeliness and Safety in Mission-Critical Systems

António Casimiro, Inês Gouveia and José Rufino

LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal,
casim@ciencias.ulisboa.pt, igouveia@lasige.di.fc.ul.pt,
jmrufino@ciencias.ulisboa.pt *

Abstract. Advances in sensor, microprocessor and communication technologies have been fostering new applications of cyber-physical systems, often involving complex interactions between distributed autonomous components and the operation in harsh or uncertain contexts. This has led to new concerns regarding performance, safety and security, while ensuring timeliness requirements are met. To conciliate uncertainty with the required predictability, hybrid system architectures have been proposed, which separate the system in two parts: one that behaves in a best-effort way, depending on the context, and another that behaves as predictably as needed, providing critical services for a safe and secure operation. In this paper we address the problem of verifying the correct provisioning of critical functions at runtime in such hybrid architectures. We consider, in particular, the KARYON hybrid architecture and its Safety Kernel. We also consider a hardware-based non-intrusive runtime verification approach, describing how it is applied to verify Safety Kernel software functions. Finally, we experimentally evaluate the performance of two distinct Safety Kernel implementations and discuss the feasibility issues to incorporate non-intrusive runtime verification.

Keywords: Real-time and embedded systems; Software architectures; Architecture hybridization; Reliability and safety; Runtime verification.

1 Introduction and Motivation

Advances in sensor, microprocessor and communication technologies have been fostering new applications of cyber-physical systems, often involving complex interactions between distributed autonomous components and the operation in harsh or uncertain contexts. A good example can be found in the automotive domain, where car makers strive to increase the autonomy of vehicles, exploiting existing technologies to make them more intelligent. While the state of the art approach consists in using information collected from local sensors to feed control loops, future cars will be connected to other cars and to the infrastructure,

* This work was partially supported by FCT, through funding of LaSIGE Research Unit, ref. UID/CEC/00408/2013. This work integrates the activities of COST Action IC1402 - Runtime Verification beyond Monitoring (ARVI), supported by COST (European Cooperation in Science and Technology).

and will cooperate for information exchange. Connectivity introduces additional security risks and, given that it is enabled by wireless networks, also introduces temporal uncertainties that conflict with real-time requirements. Additionally, processing huge amounts of incoming data will require complex processing solutions, which favor uncertainty, not predictability.

A particular challenge is to conciliate uncertainty with the required predictability, for which hybrid system architectures have been proposed. For instance, Simplex [21] considers that a control system can be composed of a controller executing in a complex subsystem, and a simple but reliable controller that is used when the complex controller malfunctions, being deployed in a separate part of the system, to be protected from potential faults in the complex subsystem.

In the scope of the KARYON project [6], we proposed the KARYON hybrid system architecture [7] to build safe cooperative systems with improved performance. This software architecture encompasses application components that execute in a complex part of the system and a Safety Kernel (SK) that, along with critical application components, should be implemented separately and should be verified to execute in a timely and reliable way. The role of the SK is to monitor the behavior of complex software components and trigger the necessary adjustments or reconfiguration actions in the complex part of the system, as needed to satisfy a set of predefined safety requirements. Note that an SK software instance will exist in each node of a distributed system (e.g., on each car) and hence the paper focuses only on a single node and not on the distribution aspects of the safety critical application or function.

In this paper we address the problem of how to verify in runtime that fundamental properties of the KARYON SK are satisfied. In fact, while it is possible to use several dependability techniques, such as the replication of software components or software verification, to enforce the required properties and raise confidence that they will be secured, these are costly and there is always a probability that, due to an accidental or even intentional fault, a property no longer holds. For example, an SK function might not complete its execution within a required temporal bound or it might produce an erroneous value. Runtime verification adds another layer of protection that is fundamental for safety assurance.

We propose a hardware-based non-intrusive runtime verification approach, which is able to detect the violation of well-defined SK properties in runtime. We describe the approach and how it is applied to verify concrete properties of the SK. We also provide experimental results that illustrate the performance of the SK implemented in two platforms, complemented with a discussion on feasibility issues relative to the incorporation of non-intrusive runtime verification.

The paper is structured as follows. Section 2 briefly reviews the KARYON hybrid architecture, describing the role of the SK. Then, Section 3 provides details on the SK design, important for explaining, in Section 4, how the runtime verification approach is applied to secure design assumptions. Relevant details of the SK implementation and a comparative evaluation of the SK operation in two different platforms is provided in Section 5. Sections 6 and 7 respectively address related work and conclude the paper.

2 Hybrid Architectures

Hybrid distributed system models and architectural hybridization [24] can be explored as a baseline design principle to address a trade-off between performance and timeliness or safety or even security. In essence, hybrid distributed system models assume that different parts of the system are characterized by different properties (for instance, each part having different timeliness properties or different integrity levels with respect to some assumed failure modes), and architectural hybridization explicitly separates system functions or components into these different parts, as needed to ensure that each component enjoys the properties provided by the part of the system in which it is allocated.

When considering the temporal domain, a system with a hybrid architecture is structured in at least two parts: one that encompasses all complex components, whose temporal behavior cannot be fully predicted or is hard to enforce, and another part that usually contains simple but critical components that execute in a predictable way. Such nice properties, like timely execution, must be enforced by design and in the implementation. For instance, dedicated hardware may be used to execute critical components, ensuring that they are temporally isolated and shielded from failures in the complex part, and that interactions between the two parts are done through a well-defined interface that preserves the properties of the part containing critical components.

The architectural hybridization concept was explored in the context of the KARYON project, which defined a generic architectural pattern for the development of sensor-based autonomous and cooperative systems [7]. This architectural pattern is shown in Figure 1.

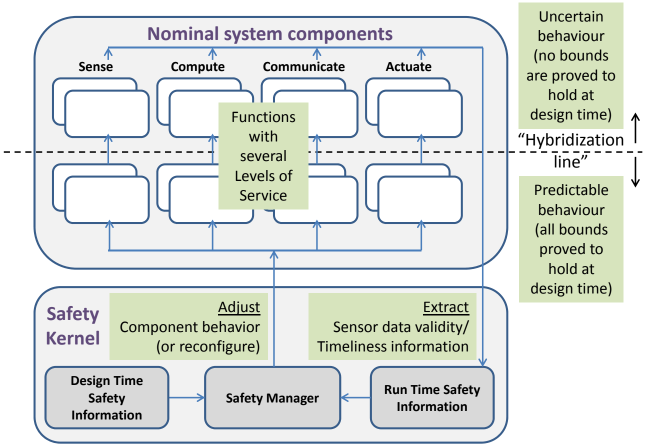
Nominal system components
Sense
Compute
Communicate
Actuate
Functions with several Levels of Service
Uncertain behaviour (no bounds are proved to hold at design time)
"Hybridization line"
Predictable behaviour (all bounds proved to hold at design time)
Safety Kernel
Adjust
Component behavior (or reconfigure)
Extract
Sensor data validity/ Timeliness information
Design Time Safety Information
Safety Manager
Run Time Safety Information

Fig. 1. The KARYON hybrid architectural pattern.

The several components that constitute the autonomous system and perform the cooperative functions are considered the *nominal system components*. These include *sensors*, *actuators*, *computation* and *communication* components. Each of these components can be used to support multiple functions. Each function can be provided with several *levels of service* (LoS), depending on the components that are being used and/or the *performance level* of each component. For instance, a function to detect obstacles ahead of a vehicle may be realized with a higher LoS if implemented using a camera sensor and an associated video processing component that is able to identify the kind of obstacle, but it may also be provided with a lower LoS by using just the information provided by a distance sensor. While it might not always be possible to execute a function at a higher LoS, namely when some needed complex component is failing to execute its function timely enough, it is assumed that it can always be executed at the lowest LoS, given that in this case it is ensured that all the involved components (considered critical ones) execute in a timely way. The hybridization line separates the system in two parts: the one where no temporal bounds can be assumed, and the predictable part, which contains critical components that are expected to execute timely and reliably (by design and implementation).

The architectural pattern is based on a Safety Kernel that is responsible for maintaining the system safe, despite the possible occurrence of faults affecting components above the hybridization line. Safety conditions are determined at design time. For each function, it is necessary to determine the safety rules that must hold to allow the function to be executed with a given LoS, that is, using a certain combination of components. For instance, the obstacle detection function can only be run at the highest LoS if the video processing component is able to timely process a video frame and provide results with good quality (which may not be possible in bad lighting conditions). The role of the Safety Kernel is hence to continuously extract information about the timeliness and quality (or validity) of sensor and processed data, use this information to verify which safety rules are satisfied, and adjust the system configuration at runtime so that all the functions are executed with the highest LoS that still secures safety.

To perform its task, the Safety Kernel includes: a Safety Manager component, a repository containing Design Time Safety Information, and a repository that is continuously updated with Runtime Safety Information. We highlight the fact that these components are located below the hybridization line. This is necessary because the Safety Kernel, as a critical component, must behave in a reliable and timely way.

3 Safety Kernel Design

Figure 2 provides an overview of the Safety Kernel functional components and the data flows between them. At startup the *eXtensible Markup Language (XML) Parser* reads a local configuration file, builds a *Safety Rules* repository and initializes *Runtime Safety Information* (RSI) structures, which will be continuously updated in runtime with the relevant safety-related information. The configuration file provides the safety rules and also *unit* definitions, expressed in XML. A unit represents a Safety Kernel input (collected data), output (adjustment data – typically a component performance level, PL) or locally calculated values (for instance, the acceptable LoS for some function). Each unit has a unique identifier that is used in the XML specification of the safety rules.

A safety rule is a boolean expression involving combinations of static values (bounds) and unit identifiers. A safety rule is meaningful for a specific LoS of some function. For instance, consider that a nominal system (e.g., an autonomous vehicle control system) is designed to perform some function F (e.g., keep a safe front distance value to any front object), and this function can be performed in two different ways (e.g., using different sensors), one way providing a higher LoS, (e.g., LoS2, allowing a smaller safety distance but requiring sensor data with high validity, possibly not achievable in some situations), and a default way providing a baseline LoS (LoS1, imposing a higher safety distance, proved to be enough even when the validity of sensor data cannot be the highest one). In this case, a safety rule would be necessary to specify the conditions for function F to be safely executed in LoS 2. If the condition (a single one, in this example) was the validity of sensor data, V_{Sens}, to be greater that 70, the safety rule would be expressed as: $F(LoS2) \rightarrow V_{Sens} > 70$.

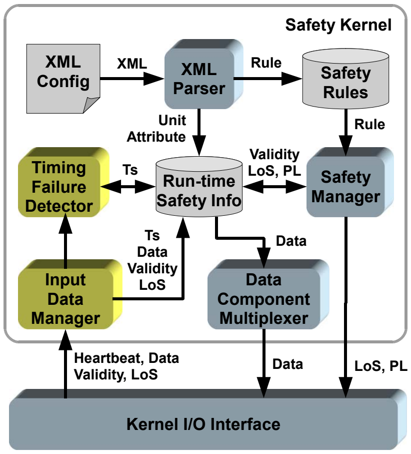

Fig. 2. Safety Kernel components.

The *Input Data Manager* receives data inputs from the external (nominal system) components and updates the RSI.

The *Timing Failure Detector* (TFD) is responsible for checking if certain data inputs have been received from external components within predefined temporal bounds. This TFD executes periodically, during each execution round of the Safety Kernel. When the TFD detects a timing failure (i.e., when some expected data is not timely produced at the Safety Kernel interface), it stores this information in the RSI unit corresponding to the untimely data. In this paper, as detailed in Section 4, we propose a design that moves into hardware a significant part of the TFD operation: the detection of timing failures.

The *Data Component Multiplexer* (DCM) selects, from two or more data inputs (collected from nominal components), one that is forwarded to its output. This is useful, for instance, when a function can be realized using one of several components that provide the same data (e.g., a front distance value), but with different timeliness or different validity. The Data Component Multiplexer selects, among the input values, the one that should be forwarded to the output (and hence nominal system), according to the permitted LoS for that function.

Finally, the *Safety Manager* is the central component as it evaluates at runtime if safety rules are satisfied given the RSI data.

4 Securing Design Assumptions Through Non-intrusive Runtime Verification

The timeliness, safety and security guarantees of Safety Kernel correct operation can be strongly enhanced through runtime verification, being of particular relevance the verification whether the design assumptions specified for the Safety Kernel are being strictly met or, somehow, have been violated.

4.1 Observer Entity

Runtime verification (RV) obtains and analyses data from the execution of a system to detect and possibly react to behaviours, either satisfying or violating a given specification. The classical approach to runtime verification implies the instrumentation of system software components, such as the Safety Kernel. Small components, which are not part of the functional system, acting as *observers*, are added to monitor and assess the state of the system in runtime.

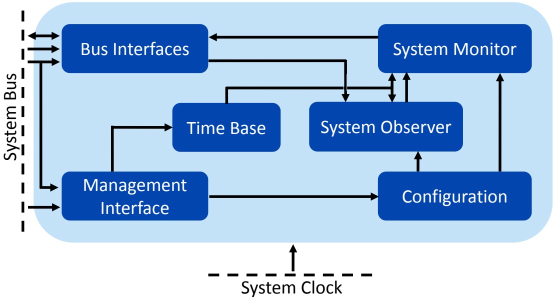

Fig. 3. Observer Entity architecture

The usage of reconfigurable logic supporting versatile FPGA-based computing platform designs enables non-intrusive approaches to runtime verification [16]. *Non-Intrusive Runtime Verification* (NIRV), provides *high flexibility*, meaning instrumentation with software-based probes is not required, although it may be used; *configurability*, which can be performed statically (offline) or dynamically, while the system is executing; *adaptability*, in the sense it is able to accommodate over time a set of different system-level, application-related and even mission-specific event observations; *independence* and *isolation*, in the sense that being supported directly in hardware, the accomplishment of runtime verification actions does not disturb nor introduce any overhead in the execution of system software components, Safety Kernel included. Similarly, the execution of software components does not ill affect the timeliness and effectiveness of runtime verification actions.

The Observer Entity is plugged to the platform where the SK software components execute, and comprises the hardware modules depicted in Figure 3: *Bus Interfaces*, capturing all physical bus activity, such as bus transfers or interrupts; *Management Interface*, enabling observer entity configuration; *Configuration*, storing the patterns of the events to be detected; the *System Observer* itself, detecting events of interest based on the set configurations; *Time Base*, which allows to time stamp events of interest, to establish its occurrence rate or to register its inter-arrival time and even to check if application-level and/or system-specific time bounds are being fulfilled; *System Monitor*, which detects possible violations to the specified system behaviour. As soon as a deviation from the specified behaviour is detected, a notification is issued.

4.2 Safety Kernel Design Assumptions and Timeliness Analysis

Let us define T_{SK} as the period of the Safety Kernel process. This process must be completed within its period, thus defining the relative deadline, D_{SK}:

$$D_{SK} = T_{SK} \tag{1}$$

The Safety Kernel process is powered by two threads: a Listener Thread, assuming the role of *Input Data Manager* (depicted in yellow in Figure 2), is activated for every incoming packet; a Periodic Thread, identified as *Timing Failure Detector* in Figure 2, runs once every Safety Kernel period. Given that $C_{listener\_thread}$ and $C_{periodic\_thread}$ represent the worst-case execution time, respectively for the Listener and Periodic threads, and that $N_{packets}$ represents the maximum total number of input packets received during a single T_{SK} period, one will have the following timing constraint:

$$N_{packets} \times C_{listener\_thread} + C_{periodic\_thread} \leq D_{SK} \tag{2}$$

As illustrated in Figure 2, there are different types of incoming packets, namely: heartbeat, data validity, multicomponent data or cooperative level of service. In the design of the Safety Kernel we assume the number of packet types is upper bounded by PKT_TYP. The worst-case execution time for a single activation of the Listener Thread, $C_{listener\_thread}$, corresponds to the longest worst-case processing time of a packet, out of the worst-case packet processing times for each packet type, $C_{pkt\_processing}(typ)$. We also take into consideration the worst-case processing time necessary to read a packet from the corresponding interface (whose location is well known and statically defined), either through (memory-mapped) I/O ports and/or network interfaces, represented by $C_{pkt\_reading}$. The maximum value of $C_{listener\_thread}$ can be expressed as:

$$C_{listener\_thread} = C_{pkt\_reading} + max_{typ=1}^{PKT\_TYP}\{C_{pkt\_processing}(typ)\} \tag{3}$$

In contrast with the Listener Thread, the Periodic Thread runs only once per SK period executing three functions in sequence and in the following order: a residual software part of the original TFD (Figure 2), that we name herein Timing Failure Detector Service Function (TFD_SF), the Safety Manager (SM) and the Data Component Multiplexer (DCM). The Periodic Thread worst-case execution time, $C_{periodic\_thread}$, is therefore given by:

$$C_{periodic\_thread} = C_{TFD\_SF} + C_{SM} + C_{DCM} \tag{4}$$

The DCM function scans the unit[1] array to find out the component data value to be forwarded and has a worst-case execution time given by C_{DCM}. The Safety Manager is a more complex function as it evaluates for each unit the safety rules and determines the new level of service or performance level. Given the number of items (e.g., number of units, number of safety rules per unit, etc.) to be processed by the Safety Manager is bounded by design, its execution time is assumed to not exceed the upper bounded given by C_{SM}. The Timing Failure Detector Service Function is much simpler: it scans the unit array to find out if there are updates (e.g., heartbeat, data validity,...) untimely received and analyses them: a minimum number of required successes and a maximum number of tolerated failures (both configured at the Safety Kernel, per input unit), have to be observed in a row to prevent instability and to steadily declare the corresponding input unit as "timely" or "non-timely", respectively. This function executes within a time that does not exceed C_{TFD_SF}, being $C_{TFD\_SF} < C_{TFD}$, the worst-case execution time of the original Timing Failure Detector entirely implemented in software.

4.3 Runtime Monitoring of Safety Kernel Operation

Let us define $t_{SK\_begin,j}$ and $t_{SK\_end,j}$, as the real-time instants where the j^{th} instance (job) of the Safety Kernel process begins and ends, respectively. Additionally, we define $n_{pkt,j}$ as the actual number of packets received within the duration of the j^{th} job of the SK process, i.e. during the interval:

$$\delta_{SK,j} = t_{SK\_end,j} - t_{SK\_begin,j} \tag{5}$$

Thus, one will have the following RV value and timing constraints:

$$\forall_{j \in \mathbb{N}} \quad 0 \leq n_{pkt,j} \leq N_{packets} \tag{6}$$

$$\forall_{j \in \mathbb{N}} \quad 0 \leq \delta_{SK,j} \leq D_{SK} \tag{7}$$

[1] A unit corresponds to a Safety Kernel information structure, concerning input (collected data), output (adjustment data) or locally calculated values. The term unit is coined from the Safety Kernel XML configuration file (§3).

Verifying that no more than $N_{packets}$ are received during each T_{SK} period, as given by Expression (6), implies: initializing an Observer Entity counting monitor with the $N_{packets}$ value each time an instance of the SK process is started; the value of the counter is decremented by one whenever a packet is received; if it reaches a value smaller than zero, a violation is signalled. Detecting when an instance of the SK process begins is achieved by configuring the address of its first instruction as an event of interest and linking it to the counting monitor.

Verifying the timeliness of an SK job implies the use of a timeliness monitor, a specialization of a counting monitor, which is initialized with the job relative deadline, as specified by Expression (7); the time counter is decrement by one at each system clock tick; if the time counter reaches zero, a timeliness violation is signalled; the time counter is stopped/restarted when an SK job is completed. Detecting when an SK job begins and when it ends is achieved by configuring, respectively, the address of its first and last instructions as events of interest, which will trigger the relevant (re)start/stop actions at the timeliness monitor.

The timing failure detection capabilities of the original Safety Kernel TFD design, described in Section 3, are herein moved to hardware and fully integrated in the Observer Entity. A timeliness monitor is instantiated for each relevant data input, being (re)started whenever a data input packet (e.g., heartbeat, data validity,...) is received by the Listener Thread. If it expires, a timing failure has been detected and it will be signalled to the Timing Failure Detection Service Function. For better integration with the software functions the signalling of timing failures is made through globally accessible memory variables.

The role of the TFD, implemented either in hardware or in software, is to detect untimely behaviours of components in the nominal system, allowing the Safety Kernel to act before any harmful effect becomes externally visible, e.g. by changing the LoS or the PL (see Figure 2). Violation of Safety Kernel design assumptions is a more severe situation, calling for some form of exception handling that hopefully will bring the system into a safe state. Since, in general, these situations were unforeseen in the design of the system, no guarantees can be provided that the adequate corrective actions (if any) are taken[2].

5 Safety Kernel implementation and evaluation

For the implementation of the Safety Kernel, a suitable hardware/software platform must be selected. The functional elements to be provided by the hardware platform include: Processing Unit, providing the computing resources; Read-Only Memory (ROM), to store the Safety Kernel software code and the safety rules; Random Access Memory (RAM), supporting the Safety Kernel execution; Input/Output (I/O) Interface, to enable the exchange of data between

[2] Most probably, there will be little to do anyway, if the design violation happens during a mission critical phase, such as the landing of a planetary probe. However, that does not necessarily imply the failure of the mission. For example: multiple (overload) alarms, occurring during the descendent flight of the first Moon landing, were advisedly discarded by the Apollo 11 lander crew.

the Safety Kernel and the nominal system components. The software platform should include fundamental real-time operating system support concerning: process/thread management and scheduling; input/output management and access, e.g. through device drivers.

5.1 Hardware platforms and software implementation

In KARYON, the fulfillment of the requirements was achieved by using a development board containing a reconfigurable logic device (FPGA), together with Intellectual Property (IP) cores from a System-on-a-Chip (SoC) library [1], mapping the functional elements into the reconfigurable logic device. The selected development board (shown on the left, in Figure 4) was a Trenz TE-0600, comprised of: Xilinx Spartan-6 FPGA; 256 MiB[3] of RAM memory; Ethernet physical interface; Flash ROM and an Secure Digital (SD) card physical interface.

The Flash ROM (not shown in Figure 4) serves as non-volatile storage for the Safety Kernel, whilst the SD Card interface supports the Safety Rules, written offline to an SD card. The FPGA supports the mapping of the controller mechanisms for these memory interfaces, together with the processing unit and Ethernet controller.

The functional elements implemented in the FPGA (shown on the right, in Figure 4) were provided by the GRLIB SoC library [1], which encompasses IP cores providing I/O functions, such as Ethernet and serial interfaces, together with the remaining components needed to implement a fully-fledged embedded computer, e.g. memory and interrupt controllers. The processing unit was implemented by the LEON3 soft-processor, a SPARCv8 architecture commonly used in avionics applications by the European space industry.

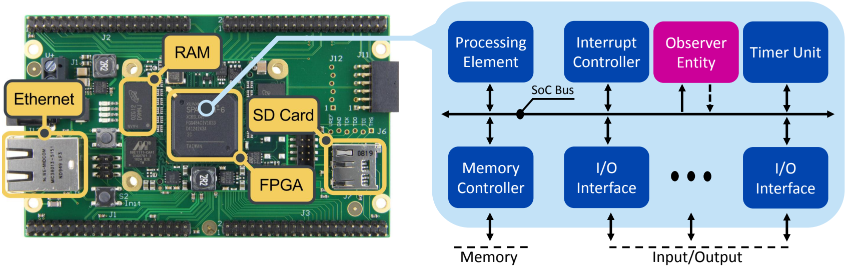

Fig. 4. Hardware platform for the Safety Kernel implementation.

Furthermore, this hardware platform is able to support all the resources required by the runtime verification techniques proposed in Section 4.3. Since the Observer Entity is essentially composed of a few counting blocks, its complexity is much lower than any other component in the FPGA (right side of Figure 4) and therefore uses only a small fraction of the occupied FPGA resources.

[3] This corresponds to the prefixes for binary multiples defined in the IEC 60027-2 standard specification [10].

As software platform, we used the RTEMS real-time operating system [15] installed on the Trenz TE-0600 hardware board. After initialization, when the configuration file is processed, two concurrent POSIX threads are used to execute the Safety Kernel functions detailed and analysed in §4.2: the Listener Thread, which handles incoming information to update the runtime safety information repository; the Periodic Thread, which is triggered every T_{SK} time units (e.g., milliseconds), where T_{SK} is the Safety Kernel execution period. This value can be changed in the XML configuration file.

To evaluate the concrete impact of using soft-processor cores, a fully-fledged software-based implementation of the Safety Kernel was deployed on an alternative platform, composed of a real-time Linux environment on a Raspberry Pi Model B Revision 2.0, with a ARM11 processor at 700Mhz [22]. Integration of non-intrusive runtime verification mechanisms was not possible in this platform, since the SoC present in the current versions of Raspberry Pi does not include the ARM CoreSight facilities [3], indispensable to secure non-intrusiveness of system observation in ARM-based platforms. The unavailability of reconfigurable logic devices on the simple Raspberry Pi platform also precludes the implementation in hardware of SK TFD functions. A Safety Kernel entirely implemented in software had to be used on the Raspberry Pi platform [22].

5.2 Performance evaluation

To properly configure the Observer Entity it is necessary to know the Safety Kernel execution period, T_{SK}. Moreover, from a practical perspective, it is also important to know if T_{SK} is sufficiently small so that the Safety Kernel can be used in a given application. In fact, this period corresponds to the maximum latency of timing failure detection and also to the time it may take for the Safety Kernel to trigger a system reconfiguration.

Therefore, we performed a set of experiments to evaluate the achievable values for T_{SK} and illustrate the feasibility of the approach. According to Expressions (1) and (2), T_{SK} depends on the worst-case execution time of two threads. The main thread involves the execution of the Timing Failure Detection (TFD) component, the Safety Manager (SM) and the Data Component Multiplexer (DCM). Given that the verification of safety rules is a complex task, the worst-case execution time of this periodic thread, $C_{periodic\_thread}$, can possibly be high. On the other hand, the Input Data Manager task is very simple, just requiring a input unit to be updated, which means that the worst-case execution time of the listener thread, $C_{listener\_thread}$, is typically much smaller. Even knowing that the listener thread wakes up several times per SK period, this number is usually limited to the number of input units. In fact, each different input unit is expected to be updated only once per SK period because there is no point in overwriting the same input unit with indications on the validity of data or on the execution timeliness of some nominal system component. The overhead of the listener thread will only become relevant in systems in which the number of different input units is high. If, for some reason, a nominal system component starts to send more packets to the SK and waking up the listener thread more times than expected, the constraint specified in Expression 6 will be violated and this will be detected by the Observer entity.

Given the above, we focused our experiments on the evaluation of the Periodic Thread response time, which in this particular case is equal to the Periodic Thread execution time, upper bounded by $C_{periodic\_thread}$.

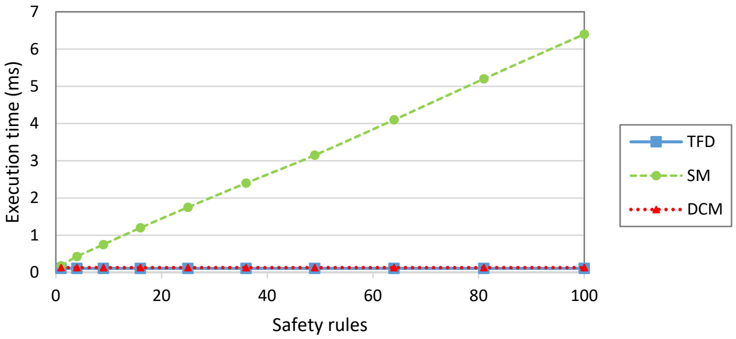

Fig. 5. Periodic thread execution time on the LEON3 soft-processor.

The first experiment was done using the SK implementation on the LEON3 soft-processor, as described in §5.1. To measure the execution time of SK components, we instrumented the SK code using start/stop timer functions provided by the underlying operating system (RTEMS or Linux), whose temporal interference on the SK execution is very small and can be neglected. Note that this instrumentation, despite intrusive, was necessary only for evaluation purposes and is fully independent from the Observer Entity, whose runtime verification mechanisms are still non-intrusive. The objective of the experiment was to determine the influence of the number of safety rules on the execution time of the periodic thread. Therefore, we created SK configuration files implying the construction of a number of safety rules varying between 1 and 100. As explained in §3, a safety rule is a Boolean expression whose value evaluates to true or false depending on input data received by the SK (through the Input Data Manager task). The safety rules we used in the evaluation involve one input unit, one output unit and a single comparison. The complexity of the safety rules evaluation algorithm stems from the need to parse a tree-like data structure, initialized at startup time and containing the input and output units, as well as the logical operations and bounds. The details of this data structure and the executed algorithm are out of the scope of this paper and can be found in [26]. For each configuration we measured the contribution of each of the three executed components (TFD, SM and DCM) for the overall execution time. The experiments were repeated 100 times and the average values were collected (the standard deviations are very small, in the order of a few microseconds, and therefore we do not show them).

The results of the first experiment are show in Figure 5. They clearly show that both the TFD and DCM components have a constant execution time, independent of the number of rules to be checked. On the other hand, the SM component execution time increases linearly with the number of safety rules. Therefore, it is possible to conclude that the SK execution time is mainly and linearly dependent on the number of safety rules, that is, on complexity of the application. However, the absolute value, which reaches 6ms for 100 safety rules, is significant. In systems requiring a reaction time of less than 60ms, at most 1000 rules would be acceptable, which seems limited. The reason for such high execution time is fundamentally due to the fact that the SK is running on a soft-processor infrastructure.

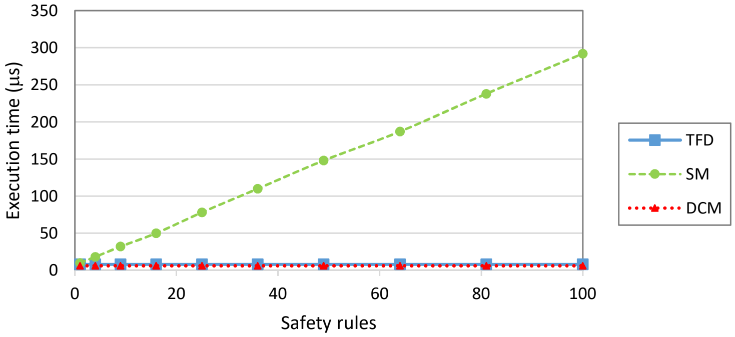

Fig. 6. Periodic thread execution time on a Raspberry PI.

To understand the concrete impact of using a soft-processor, we performed a second experiment by deploying the SK on a real-time Linux/Raspberry Pi platform. The same experiments were performed and yielded the results shown in Figure 6.

The most important observation is the significant reduction of the execution time, as expected. Instead of 6ms, processing 100 safety rules takes no more than 300μs, which is 20 times less. The approach seems thus feasible for most applications, provided that a reasonably good processor is used.

5.3 Effectiveness and feasibility analysis

The Safety Manager has, in general, a worst-case execution time, T_{SM}, which largely exceeds those of the Timing Failure Detector, either of the entirely software-based solution, T_{TFD}, or of the hardware/software co-design introduced in section 4, T_{TFD_SF}. Since, $T_{TFD\_SF} < T_{TFD} << T_{SM}$, the performance improvement due to a smaller T_{TFD_SF} value is not significant in terms of the overall Safety Kernel operation. Methods to reduce the value of T_{SM}, allowing a significant performance improvement, will be addressed in future work.

At this point, the main benefit provided by the non-intrusive runtime verification mechanisms is to secure the Safety Kernel design assumptions. Instead of resorting to classical code instrumentation, which is inherently intrusive, our approach relies on independent, isolated and non-intrusive runtime verification mechanisms, easily integrated in reconfigurable logic supporting soft-processors (e.g., LEON3), such as the Trenz TE-0600 platform (see §5.1). Integration of non-intrusive runtime verification mechanisms in platforms based on ARM processors is dependent on the availability of ARM CoreSight facilities [3].

Detecting a violation of SK design assumptions may significantly contribute to enhance the overall system dependability. For some usages, a special-purpose exception handler could be programmed within the SK context to activate existing safeguard functions, e.g. for the safe stop of a terrestrial/maritime unmanned autonomous vehicle. In general, such functions may not exist (cf. §4.3).

6 Related Work

A novel perspective on distributed systems' architecture was settled by the notion of hybridization in [25] and [23]. The concept of architectural hybridization and its diverse advantages were further discussed in [24]. System parts with distinct synchronism [25] or security [8] properties can take advantage of hybrid distributed system model approaches. Hybrid system modeling has also been previously applied to autonomous control systems [2]. The hybrid nature of systems was also acknowledged in [14], which developed a component-based generic platform for embedded real-time systems.

Both offline and online runtime verification (RV) approaches have been previously studied, with online RV receiving increased attention due to its many benefits regarding safety and performance [4]. Furthermore, non-intrusive runtime monitoring has been previously applied in embedded systems [27, 17] and, more specifically, in safety critical environments [11], presenting an RV architecture for monitoring safety critical embedded systems using an external bus monitor connected to the target system. A novel System Health Management technique was introduced in [18] which empowers both real-time assessment of the system status with respect to temporal-logic-based specifications and supports statistical reasoning to estimate its health at runtime. Configurable non-intrusive event-based frameworks for runtime monitoring have been developed within the embedded systems' scope [13], employing a minimally intrusive method for dynamic monitoring. Additionally, the RV concept has been applied to cyber-physical systems [28], autonomous systems [5], avionic systems [20, 19] and to an AUTOSAR-like real-time operating system, aiming the automotive domain [9]. [12] describes a runtime monitoring approach for autonomous vehicle systems requiring no code instrumentation by observing the network state.

7 Conclusion

This paper addressed the problem of hardware-based non-intrusive runtime verification, considering its application on a system with a hybrid architecture. Hybridization allows separating the system in at least two parts, making strong assumptions (on the temporal and/or security domains) only for one of the parts, typically a small one. It is thus important not only to verify in design time that these strong assumptions are effectively satisfied, but also to verify them in runtime, particularly when the operational conditions cannot be fully anticipated.

We described an approach for non-intrusive runtime verification and explained how it is applied in a concrete case: to verify a set of assumptions underlying the design of a Safety Kernel, also described in the paper. The approach was used to verify timing assumptions and also assumptions on the maximum number of events occurring in a time interval.

Finally, the paper also provided experimental results to illustrate the performance that might be expected from two implementations of a Safety Kernel: one running on a soft-processor and another running on a real ARM processor. The results show that with a hardware processor it is possible to use a Safety Kernel in complex applications. On the other hand, we described some feasibility constraints for applying our verification approach on ARM processors. We plan to address these constraints in future work in order to take full advantage of the proposed non-intrusive verification approach.

References

1. Aeroflex Gaisler A.B., Goteborg, Sweden: GRLIB IP Library User's Manual (Apr 2014)
2. Antsaklis, P.J., Stiver, J.A., Lemmon, M.: Hybrid system modeling and autonomous control systems. In: Hybrid systems, pp. 366–392. No. 736 in LNCS, Springer, Berlin, Heidelberg (1993)
3. ARM: ARM CoreSight Architecture Specification, 2.0 edn. (Sep 2013)
4. Backasch, R., Hochberger, C., Weiss, A., Leucker, M., Lasslop, R.: Runtime verification for multicore SoC with high-quality trace data. ACM Transactions on Design Automation of Electronic Systems (TODAES) 18(2), 18 (Mar 2013)
5. Callow, G., Watson, G., Kalawsky, R.: System modelling for run-time verification and validation of autonomous systems. In: Proc. 5th Int. Conference on System of Systems Engineering. pp. 1–7. Loughborough, UK (Jun 2010)
6. Casimiro, A., Kaiser, J., Schiller, E.M., Costa, P., Parizi, J., Johansson, R., Librino, R.: The KARYON Project: Predictable and safe coordination in cooperative vehicular systems. In: Proc. 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W). pp. 1–12. IEEE (2013)
7. Casimiro, A., Rufino, J., Pinto, R.C., Vial, E., Schiller, E.M., Morales-Ponce, O., Petig, T.: A kernel-based architecture for safe cooperative vehicular functions. In: Proc. 9th IEEE International Symposium on Industrial Embedded Systems (SIES). pp. 228–237 (June 2014)
8. Correia, M., Veríssimo, P., Neves, N.F.: The design of a COTS real-time distributed security kernel. In: 4th European Dependable Computing Conference. pp. 234–252. Springer, Toulouse, France (Oct 2002)
9. Cotard, S., Faucou, S., Bechennec, J.L., Queudet, A., Trinquet, Y.: A data flow monitoring service based on runtime verification for AUTOSAR. In: Proc. 14th Int. Conf. on High Performance Computing and Communications. IEEE, Liverpol, UK (Jun 2012)
10. IEC Standards: IEC 60027-2: Letter symbols to be used in electrical technology Part 2: telecommunications and electronics (Aug 2005)
11. Kane, A.: Runtime Monitoring for Safety-Critical Embedded Systems. Ph.D. thesis, Carnegie Mellon University, USA (Feb 2015)

12. Kane, A., Chowdhury, O., Datta, A., Koopman, P.: A case study on runtime monitoring of an autonomous research vehicle (ARV) system. In: Proc. 15th Int. Conf. on Runtime Verification. pp. 102–117. Vienna, Austria (Sep 2015)
13. Lee, J.C., Lysecky, R.: System-level observation framework for non-intrusive runtime monitoring of embedded systems. ACM Transactions on Design Automation of Electronic Systems 20(42) (2015)
14. Obermaisser, R., Kopetz, H.: Genesys: A candidate for an ARTEMIS cross-domain reference architecture for embedded systems. 2009 (Sep 2011)
15. On-Line Applications Research Corporation: RTEMS C User's Guide, 4.9.4 edn. (2010)
16. Pinto, R.C., Rufino, J.: Towards non-invasive run-time verification of real-time systems. In: Proc. 26th Euromicro Conf. on Real-Time Systems - WIP Session. pp. 25–28. Madrid, Spain (Jul 2014)
17. Reinbacher, T., Fugger, M., Brauer, J.: Runtime verification of embedded real-time systems. Formal Methods in System Design 24(3), 203–239 (2014)
18. Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 357–372. Springer, Grenoble, France (Apr 2014)
19. Rufino, J., Gouveia, I.: Timeliness runtime verification and adaptation in avionic systems. In: Proc. 12th workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT). Toulouse, France (Jul 2016)
20. Rufino, J.: Towards integration of adaptability and non-intrusive runtime verification in avionic systems. SIGBED Review 13(1) (Jan 2016), (Special Issue on 5th Embedded Operating Systems Workshop)
21. Sha, L.: Using simplicity to control complexity. IEEE Software 18(4), 20–28 (Jul/Aug 2001)
22. Upton, E., Halfacree, G.: Raspberry Pi User Guide. Wiley (2012)
23. Verissimo, P.: Uncertainty and predictability: Can they be reconciled? In: Future Directions in Distributed Computing, LNCS, vol. 2584, pp. 108–113. Springer-verlag, Berlin Heidelberg (2003)
24. Verissimo, P.: Travelling through wormholes: a new look at distributed systems models. SIGACT News, vol. 37, no. 1, pages 66-81, 2006. (Mar 2006)
25. Verissimo, P., Casimiro, A.: The timely computing base model and architecture. Computers, IEEE Transactions on 51(8), 916–930 (2002)
26. Vial, E., Casimiro, A.: Evaluation of safety rules in a safety kernel-based architecture. In: et al., A.B. (ed.) Workshop on Architecting Safety in Collaborative Mobile Systems (ASCoMS'14), Proceedings of the 2014 international conference on Computer Safety, Reliability, and Security. pp. 27–35. No. 8696 in LNCS, Springer-Verlag, Florence, Italy (Sep 2014)
27. Watterson, C., Heffernan, D.: Runtime verification and monitoring of embedded systems. Software, IET 1(5) (Oct 2007)
28. Zheng, X., Julien, C., Podorozhny, R., Cassez, F.: BraceAssertion: Runtime verification of cyber-physical systems. In: Proc. 15th IEEE Real-Time and Embedded Tech. and Applications Symposium. pp. 298–306 (Oct 2015)