Non-intrusive Observation and Runtime Verification of Avionic Systems*

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

Abstract

Unmanned autonomous systems (UAS) avionics call for advanced computing system architectures fulfilling strict size, weight and power consumption (SWaP) requisites. The AIR (ARINC 653 in Space Real-Time Operating System) defines a partitioned environment for the development and execution of aerospace applications, preserving application timing and safety requisites.

This paper intensively explores the potential of non-intrusive runtime verification (NIRV) mechanisms, currently being included in AIR, to the overall improvement of system safety.

1 Introduction and Motivation

Avionic systems have strict safety and timeliness requirements as well as strong size, weight and power consumption (SWaP) constraints. Modern unmanned autonomous systems (UAS) avionics follow the civil aviation trend of transitioning from federated architectures to Integrated Modular Avionics (IMA) [1] and resort to the use of partitioning.

Partitioning implement the logical separation of applications in criticality domains, that we named partitions, and allow hosting both avionic and payload functions in the same computational infrastructure [2, 3].

However, partitioned architectures in general, and those designed using AIR (ARINC 653 in Space Real-Time Operating System) [4], in particular, tend to have their complexity and may largely benefit of their combination with a runtime verification and monitoring infrastructure [5].

This paper explains how fundamental runtime verification (RV) mechanisms can be combined with advanced time- and space-partitioned (TSP) systems. To reduce the temporal overhead of such mechanisms in the operation of onboard systems an innovative non-intrusive design approach is followed.

The paper is organized as follows. Section 2 describes the non-intrusive RV features being introduced while Section 3 presents the AIR architecture for TSP systems. Section 4 describes how to integrate RV mechanisms with the AIR architecture and Section 5 performs their evaluation. Section 6 describes the related work and, finally, Section 7 issues concluding remarks and future research directions.

*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).

2 Mechanisms for Non-intrusive Observation and Runtime Verification

Runtime verification obtains and analyses data from the execution of a system to detect and possibly react to behaviours, either satisfying or violating the system specification. Runtime verification implies that 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.

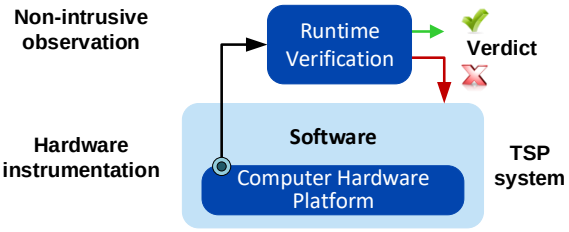

Figure 1: Non-intrusive observer and runtime verification.

The usage of reconfigurable logic supporting versatile platform designs (e.g., soft-processors), as depicted in Figure 1, enables innovative approaches to RV [6]. In the context of TSP systems: the computer hardware platform is instrumented with *non-intrusive* observers; the *runtime verification* is secured by an independent hardware module, with no system actions (unless there is an error).

An enhanced AIR architecture uses an *AIR Observer and Monitor* (AOM) featuring: *non-intrusiveness*, meaning system operation is not adversely affected and code instrumentation with RV probes is not required; *configurable*, being able to accommodate different event observations.

The AOM hardware is plugged to the platform where the AIR software components execute, and comprises the modules depicted in Figure 2: **bus interfaces**, capturing all physical bus activity, such as system bus and cache bus transfers or interrupts; **management interface**, enabling AOM configuration; **configuration**, storing the patterns of the events to be detected; **observer**, detecting events of interest based on the registered configurations and **monitor**, performing the required runtime verification actions.

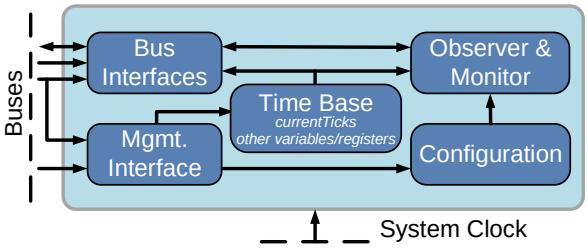

Figure 2: AIR Observer and Monitor architecture.

Though RV concepts can be applied to both time and space partitioning, this paper is restricted to temporal issues. Thus, it is assumed that a **robust time base**[1] accounts for, in the AOM hardware (Figure 2), the number of POS-level clock ticks elapsed so far, to which AIR components have access, through the read only $currentTicks$ variable/register (used in Algorithm 1). Other variable/registers may need to be stored within the scope of the AOM hardware.

3 AIR Technology for TSP systems

The AIR design aims at providing high levels of flexibility, hardware- and OS-independence, easy integration and independent component verification, validation and certification [4]. The AIR architecture is depicted in Figure 3.

The *AIR Partition Management Kernel (PMK)* is a core software layer, enforcing robust TSP properties, together with partition scheduling and dispatching, low-level interrupt management, and interpartition communication support. Robust TSP implies that the execution of functions in one partition does not affect other partitions' timeliness and that separated addressing spaces are assigned to different partitions.

Each partition can host a different OS (the partition operating system, POS), which in turn can be either a real-time operating system (RTOS) or a generic non-real-time one. The *AIR POS Adaptation Layer* (PAL) encapsulates the POS of each partition, providing an adequate POS-independent interface.

The *Portable Application Executive* (APEX) *interface* [7] provides a standard programming interface derived from the ARINC 653 specification [1], with the possibility of being subsetted and/or adding specific functional extensions for certain partitions [8].

The architecture of Figure 3 also includes the AOM hardware module that we will intensively exploit in our design.

4 Integrating Non-intrusive Observation and Runtime Verification

The integration of RV features in the AIR architecture is, in essence, concerned with the operation of the AIR Partition Scheduler and Dispatcher and uses a dual approach:

- operation enforced in hardware, either totally or with some degree of assistance from software components, being the RV actions performed in software, being this kind of action only seldom used;

[1]The design and engineering of AIR robust timers is out of the scope of this paper. It will be addressed in a future work.

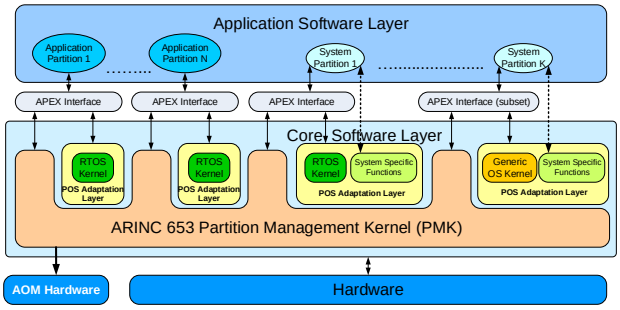

Figure 3: AIR architecture with AOM hardware.

- operation achieved through the execution of software components, with RV actions enforced in hardware, the normal operating behaviour.

4.1 Partition scheduling

The original ARINC 653 notion of a single fixed Partition Scheduling Table (PST) [1], defined offline, is limited in terms of timeliness, as well as safety and fault-tolerance control. To address this primary limitation, the AIR design incorporates the notion of *mode-based partition schedules*, inspired by the optional service defined within the scope of ARINC 653 Part 2 specification [9].

The system can now be configured with multiple PSTs, which may differ in terms of their Major Time Frame (MTF) duration. The different PSTs may specify which partitions are scheduled on each mission phase, and of how much processor time is assigned to them [4], as shown in Figure 4. The system can then switch between these PSTs; a PST switch request is only effectively granted at the end of the ongoing MTF [4].

4.2 Mode-based schedules

The AIR RV architecture uses an hardware-assisted approach for selecting the partition scheduling switch instants, which are programmed at the AOM, whenever a partition is dispatched: the next *partition preemption point* is inserted in the AOM configuration; when this instant is reached, an AOM's hardware exception triggers the execution of Algorithm 1.

The RV actions of **Algorithm 1** check, from the active PST, if the current instant is a partition preemption point (line 3). If that is not the case, a severe system level error has occurred and the Health Monitor is notified (line 4) to handle the situation. The AIR Health Monitor is a component, not represented in Figure 3, that aims to contain faults within their domains of occurrence, to provide the corresponding error handling capabilities and that it spreads throughout virtually all of the AIR architectural components. The remaining lines (6-12) implement the partition switch actions of [4], checking

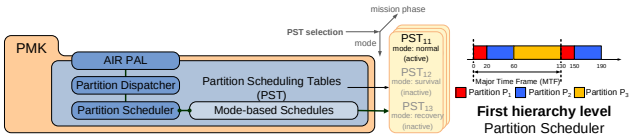

Figure 4: Partition scheduling featuring mode-based schedules.

Algorithm 1 AIR Partition Scheduler, with Runtime Verification featuring mode-based schedules

```
1: ▷ Entered upon exception: partition preemption point detected
2: ▷ Runtime verification actions
3: if schedules_currentSchedule.table_tableIterator.tick ≠
       (currentTicks − lastScheduleSwitch) mod
       schedules_currentSchedule.mtf then
4:     HEALTHMONITOR(activePartition)
5: else  ▷ Partition switch actions
6:     if currentSchedule ≠ nextSchedule ∧
           (currentTicks − lastScheduleSwitch) mod
           schedules_currentSchedule.mtf = 0 then
7:         currentSchedule ← nextSchedule
8:         lastScheduleSwitch ← currentTicks
9:         tableIterator ← 0
10:    end if
11:    heirPartition ←
       schedules_currentSchedule.table_tableIterator.partition
12:    tableIterator ← (tableIterator + 1) mod
       schedules_currentSchedule.numberPartitionPreemptionPoints
13: end if
```

Algorithm 2 AIR Partition Dispatcher, with Runtime Verification updating partition preemption points

```
1: ▷ Entered from the AIR Partition Scheduler after partition switch actions
2: SAVECONTEXT(activePartition.context)
3: activePartition.lastTick ← currentTicks − 1
4: elapsedTicks ← currentTicks − heirPartition.lastTick
5: activePartition ← heirPartition
6: REPLACEPREEMPTIONPOINT(heirPartition.tick)
7: RESTORECONTEXT(heirPartition.context)
```

(line 6) if there is a pending scheduling switch to be applied and the current instant is the end of the MTF. If these conditions apply, a different PST will be used henceforth (line 7). The processing resources are assigned to the heir partition, obtained (line 11) from the PST in use. The Partition Scheduler is set (line 12) to access the heir partition parameters.

4.3 Partition dispatching

The execution is followed by the AIR RV Partition Dispatcher specified in **Algorithm 2**. Two significant differences do exist from the software-based approach of [4]: elapsed clock ticks settings is no longer needed because the partition dispatcher is always invoked after a partition switch; insertion of the next partition preemption point in the hardware-assisted AOM configuration (line 6). The remaining actions in Algorithm 2 are related to saving and restoring the execution context (lines 2 and 7) and evaluation of the elapsed clock ticks (line 4).

4.4 Observation of application components

Besides the AIR RV Partition Scheduler and Dispatcher, two fundamental parts of our system, one dedicate our attention to the monitoring of other components, such as the applications. Through the use of the AOM module, observation and monitoring continues to be non-intrusive. This is done through **Algorithm 3**, the AIR Event Observer.

The AOM observes the Bus, compares (line 6) the transfer operations $Bus.trf$ with a configured set of observation points, $Config$. Upon match, it sends a piece of information to the external system (line 9). This piece of information is an $event$, being comprised of: the time-stamp of the occurrence; the id of the event, specified in the configuration (lines 7-8). The $numTick$ value (line 4) is incremented at every system clock tick, and used as the event time-stamp.

Algorithm 3 AIR Event Observer

```
1: ▷ Input: Clock - system_clock_tick; Bus - raw event
2: ▷ Output: Event - event
3: for system_clock_tick do
4:     numTicks ← numTicks + 1
5:     if newEvent(Bus) then
6:         if ∃ id ∈ Config : Config[id] = Bus.trf then
7:             event.time ← numTicks
8:             event.id ← id
9:             outputEvent(event)
10:        end if
11:    end if
12: end for
```

5 Evaluation: analysis and discussion

One relevant metric for code complexity is its size, in lines of source code. The standardized accounting method one employ is the *logical source lines of code (logical SLOC)* metric of the Unified CodeCount tool [11]. The C implementation of fundamental AIR components, such as the AIR Partition Scheduler and Dispatcher, is assessed in Table 1, which shows its logical SLOC count along with the entity instantiating the component, and implicitly, the instantiation frequency. The data show a reduction of code complexity.

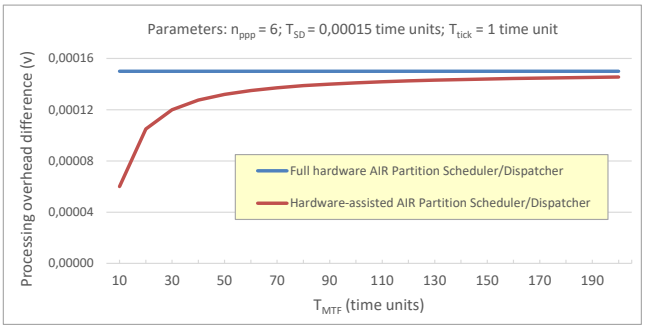

Figure 5: Analysis of processing time overheads.

With respect timing issues, comparing the normalised processing time overheads of AIR Partition Scheduler and Dispatcher ($\mathcal{T}_{SD}$), in the software-based and hardware-assisted approaches, along a full normalised MTF period ($\mathcal{T}_{MTF}$):

$$v \approx \frac{\mathcal{T}_{SD\_Soft}}{\mathcal{T}_{sys\_tick}} - \frac{\mathcal{T}_{SD\_Hard}}{\mathcal{T}_{MTF}} \cdot n_{ppp} \quad (1)$$

where, n_{ppp} is the number of partition preemption points in the MTF and $\mathcal{T}_{sys\_tick}$ is the normalised POS-level clock tick. The normalisation of timing parameters in Figure 5 take the experimental values $\mathcal{T}_{SD\_Soft} = 150\ ns$ and $\mathcal{T}_{sys\_tick} = 1\ ms$ as references, making $\mathcal{T}_{SD\_Hard} \approx \mathcal{T}_{SD\_Soft}$ for hardware-assisted and $\mathcal{T}_{SD\_Hard} = 0$ for a full hardware implementation of the AIR Partition Scheduler/Dispatcher [12].

To exemplify the use of AIR AOM hardware in the observation/monitoring of several events, one considerer the Attitude and Orbit Control Subsystem (AOCS) function of a Low Earth Orbit (LEO) satellite. The Cartesian coordinates are used to evaluate the satellite position:

$$(x - u_x)^2 + (y - u_y)^2 + (z - u_z)^2 \leq (\delta_d)^2 \quad (2)$$

where, (x, y, z) are the real position of the satellite and (u_x, u_y, u_z) are the specified satellite position; the value δ_d defines a specified maximum distance deviation.

Table 1: Logical SLOC metrics and instantiation entities for fundamental AIR software components.

	Logical SLOC	Instantiation
Software-based AIR Partition Scheduler (specified an analysed in [4, 10])	13	POS-level clock tick
Software-based AIR Dispatcher (specified an analysed in [4, 10])	10	POS-level clock tick
Hardware-assisted AIR RV Partition Scheduler (specified in Algorithm 1)	12	partition preemption point
Hardware-assisted AIR RV Dispatcher (specified in Algorithm 2)	8	partition preemption point

The real position of the satellite is read and compared with the specified position. This difference should be kept below a given and specified threshold. If a violation occurs, such an event will be signalled to the AIR Health Monitor.

The synthesis of a monitor can be ensured with TeSSLa [13], a Temporal Stream-based Specification Language, which is specially designed for specifying correct program executions.

6 Related Work

Approaches to flexible scheduling in TSP systems are restricted to the mode-based scheduling of the commercial Wind River VxWorks 653 product [14]. Alternatives to TSP/IMA are compared in [15], which includes recommendations for adaptation of IMA-like solutions. Emergence of non-intrusive runtime verification techniques for embedded systems in general is addressed in [16, 17], while its applicability to complex safety-critical systems is presented in [18]. However, no previous work have applied such techniques to the realm of TSP systems.

7 Conclusion

This paper addressed how mechanisms providing support to the AIR architecture for time- and space-partitioned systems can be designed and engineered. The usage of a non-intrusive *AIR Observer and Monitor* allows not only the monitoring of fundamental AIR components but also of generic events. Non-intrusive runtime verification is a relevant contribution with respect to verification, validation and certification efforts of TSP systems that will be extended in future research.

References

[1] AEEC (Airlines Electronic Engineering Committee), *Avionics Application Software Standard Interface, Part 1 - Required Services*, Mar. 2006.

[2] TSP Working Group, "Avionics time and space partitioning user needs," Technical Note TEC-SW/09-247/JW, ESA, Aug. 2009.

[3] J. Rushby, "Partitioning in avionics architectures: Requirements, mechanisms and assurance," Tech. Rep. NASA CR-1999-209347, SRI International, June 1999.

[4] J. Rufino, J. Craveiro, and P. Verissimo, "Architecting robustness and timeliness in a new generation of aerospace systems," in *Architecting Dependable Systems VII*, vol. 6420 of *LNCS*, Springer, 2010.

[5] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebric Programming*, vol. 78, pp. 293–303, May-Jun 2009.

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

[7] S. Santos, J. Rufino, T. Schoofs, C. Tatibana, and J. Windsor, "A portable ARINC 653 standard interface," in *Proc. 27th Digital Avionics Systems Conf.*, (St. Paul, MN, USA), Oct. 2008.

[8] J. Rosa, J. P. Craveiro, and J. Rufino, "Safe online reconfiguration of time- and space-partitioned systems," in *Proc. 9th IEEE Int. Conf. on Industrial Informatics (INDIN 2011)*, (Caparica, Lisbon, Portugal), July 2011.

[9] AEEC (Airlines Electronic Engineering Committee), *Avionics Application Software Standard Interface, Part 2 - Extended Services*, Dec. 2008.

[10] J. P. Craveiro and J. Rufino, "Adaptability support in time- and space-partitioned aerospace systems," in *Proc. 2nd Int. Conf. on Adaptive and Self-adaptive Systems and Applications*, (Lisbon, Portugal), Nov. 2010.

[11] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, "A SLOC counting standard," in *The 22nd Int. Ann. Forum on COCOMO and Systems/Software Cost Modelling*, (Los Angeles, USA), 2007.

[12] J. Rufino, "Towards integration of adaptability and non-intrusive runtime verification in avionic systems," *ACM SIGBED Review*, vol. 13, Jan. 2016.

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

[14] Wind River, "Wind River VxWorks 653 Platform 2.4 and 2.5," 2015.

[15] B. Ford, P. Bull, A. Grigg, L. Guan, and I. Phillips, "Adaptive architectures for future highly dependable, real-time systems," in *Proc. 7th Ann. Conf. on Systems Engineering Research*, (Loughborough, UK), Apr. 2009.

[16] C. Watterson and D. Heffernan, "Runtime verification and monitoring of embedded systems," *Software, IET*, vol. 1, pp. 172–179, October 2007.

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

[18] A. Kane, *Runtime Monitoring for Safety-Critical Embedded Systems*. PhD thesis, Carnegie Mellon University, USA, Feb. 2015.