Timeliness Runtime Verification and Adaptation in Avionic Systems

José Rufino and Inês Gouveia
LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
jmrufino@ciencias.ulisboa.pt, igouveia@lasige.di.fc.ul.pt

***Abstract*—Unmanned autonomous systems (UAS) avionics call for advanced computing system architectures fulfilling strict size, weight and power consumption (SWaP) requisites, decreasing the vehicle cost and ensuring the overall system dependability. The AIR (ARINC 653 in Space Real-Time Operating System) architecture defines a partitioned environment for aerospace applications, following the notion of time and space partitioning (TSP), aiming to preserve the highly demanding application timing and safety requirements.**

In addition to expected changes in the vehicle configuration, which may naturally vary according to the mission's progress and its phases, a vehicle may be exposed to unforeseeable events (e.g., environmental) and to failures. Thus, vehicle survivability requires advanced adaptability and reconfigurability features, to be supported in the AIR architecture. Adaptation in the presence of hazards may largely benefit from the potential of non-intrusive runtime verification (RV) mechanisms, currently being included in AIR. Although this paper focuses on system level (timeliness) monitoring and adaptation, similar approaches and methods may be taken with respect to application/mission adaptation.

I. 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.

Partitioned architectures implement the logical separation of applications in criticality domains, named partitions, and allow hosting both avionic and payload functions in the same computational infrastructure, thus fulfilling both SWaP and safety/timeliness requirements [2]. Avionic functions are related with vehicle control and typically include: Attitude and Orbit Control Subsystem (AOCS) or Guidance, Navigation and Control (GNC); Onboard Data Handling (OBDH); Telemetry, Tracking and Command (TTC); Fault Detection, Isolation and Recovery (FDIR). On the other hand, payload functions are strictly related with the mission's purpose. Partitioning implies that each one of those functions is hosted in a different partition.

The notion of temporal and spatial partitioning (TSP) means that the execution of functions in one partition does not affect other partitions' timeliness and that dedicated addressing spaces are assigned to different partitions [3]. The design and development of AIR (ARINC 653 in Space Real-Time Operating System) has been motivated by the interest in applying TSP concepts to the aerospace domain [4]. However, TSP concepts can be applied to a broader set of applications such as, planetary exploration, automotive, underwater rovers, and aquatic/aerial drones. The case for low-cost drones, commonly available as radio-controlled gadgets, with little or no provisions of safety guarantees, is specially sensitive.

Usually, an UAS mission goes through multiple phases (e.g., takeoff, flight, approach, exploration, flight back, landing). Adaptation to changing temporal requirements as the mission progresses, throughout its phases, is of great importance. Furthermore, adaptation to unplanned circumstances, such as unforeseeable external events and internal failures, is mandatory for vehicle and mission's survivability [5]. The design of AIR Technology already includes mechanisms of support for adaptation and reconfiguration [6]. Nevertheless, given the high complexity of UAS functions, modern avionic systems may largely benefit from the verification in runtime whether or not the system/mission parameters are in conformity with the planned specification.

This paper addresses how innovative non-intrusive runtime verification (RV) capabilities, specially designed for time- and space-partitioned systems, may enable the design and implementation of advanced timeliness adaptation mechanisms, which allow to reduce the temporal overhead of such mechanisms in the operation of onboard systems.

The paper is organized as follows. Section II introduces the AIR architecture for TSP systems. Section III describes the non-intrusive RV features being introduced in the AIR architecture. Section IV details the new adaptability features of AIR. Section V discusses the integration of those features in the AIR architecture and performs its analysis. Section VI describes the related work and, finally, Section VII presents some concluding remarks and future research directions.

II. AIR Technology for TSP systems

The AIR Technology evolved from a proof of feasibility for adding ARINC 653 functional support to the Real-Time Executive for Multiprocessor Systems (RTEMS) to a multi-OS (operating system) TSP architecture [4]. The AIR modular design aims at high levels of flexibility, hardware- and OS-independence (through encapsulation), easy integration and independent component verification, validation and certification.

This work was partially supported by FCT, through LaSIGE Strategic Project 2015-2017, UID/CEC/00408/2013. This work integrates the activities of COST Action IC1402 - Runtime Verification beyond Monitoring (ARVI).

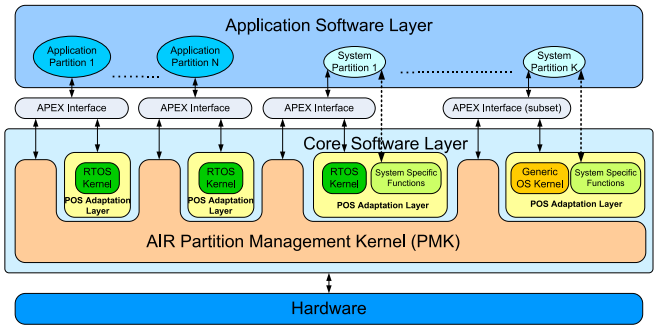

Fig. 1. AIR architecture for TSP systems

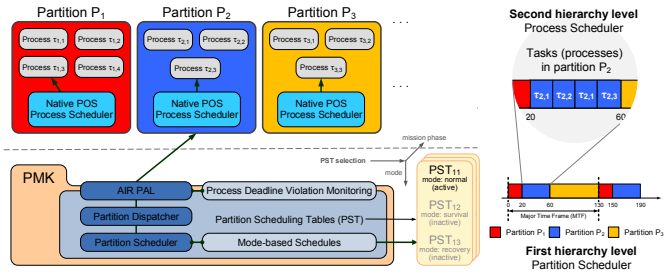

Fig. 2. Two-level hierarchical scheduling with partition scheduling featuring timeliness adaptation through mode-based schedules

A. System architecture

The AIR modular architecture is pictured in Figure 1. The *AIR Partition Management Kernel (PMK)* is the basis of a core software layer, enforcing robust TSP properties and providing support to fundamental mechanisms such as partition scheduling and dispatching, low-level interrupt management, interpartition communication facilities and encapsulation of special-purpose hardware resources. Temporal partitioning ensures that the real-time requisites of the different functions executing in each partition are guaranteed. Spatial partitioning relies on having dedicated addressing spaces for the functions executing on 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 to the surrounding components.

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, on a system-level and/or on a per-partition basis [8].

The organization of vehicle functions in different partitions requires interpartition communication services, since a function hosted in a partition may need to exchange information with other partitions. Interpartition communication consists of the authorized transfer of information between partitions without violating neither spatial separation constrains nor information security properties [3], [4], [9].

B. Two-level scheduling

The AIR technology employs a two-level scheduling scheme, as illustrated in Figure 2. The first level corresponds to partition scheduling and the second level to process scheduling. Partitions are scheduled on a cyclic basis, through the partition scheduling and dispatching components (Figure 2), according to a partition scheduling table (PST) repeating over a major time frame (MTF). The PST assigns execution time windows to partitions. Inside each partition's time windows, its processes compete for processing resources according to the POS's native process scheduler.

This paper proposes an evolution of the AIR two-level hierarchical scheduling towards a highly effective short-term adaptation of timeliness parameters to the mission phase and/or to environmental changes, through a proficient use of mode-based schedules, as highlighted in Figure 2.

C. Health monitoring and event handling

The AIR architecture incorporates *Health Monitor* (HM) functions that spread throughout virtually all of the AIR architectural components, aiming to contain faults within their domains of occurrence.

At system-level, HM functions monitor the correctness of fundamental AIR system components. In the event of an error, handling is performed through fully integrated HM event handlers (Figure 3). For example: system-level timeliness (e.g., partition scheduling) is verified at runtime, with a contingency signalling of timing errors through low-level event handlers.

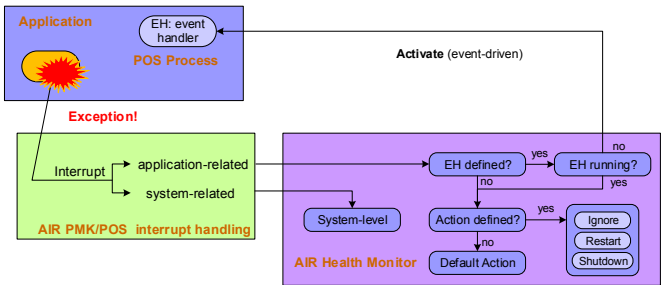

Fig. 3. System-level and application-level health monitoring

At application-level (Figure 1), which comprises both avionics and payload functions, HM functions aim to enforce overall correctness and to prevent the ill-effects of process and/or partition level errors, occurring at one partition, from propagating to the remaining partitions.

The runtime verification of application correctness is deeply dependent on the application itself. Detection of deviations from a given application/mission specification and handling of abnormal situations must be performed by special-purpose event handlers, provided by the application programmer and/or by the system integrator, as shown in the diagram of Figure 3. Only some specific aspects of application correctness may be verified at system-level (e.g., the monitoring of violations to registered process deadlines, pointed out in Figure 2).

In any case, the actions to be performed in the event of errors are cast into appropriate event handlers. These may comprise adaptability features such as the redefinition of timing and control parameters. If no handler is provided, a response action defined by the partition's HM ARINC 653 configuration table is executed, as shown in Figure 3. The design of AIR allows HM handlers to simply replace existing exception handlers or to be added to existing ones, in pre- and/or post-processing modes.

III. TSP-oriented Non-intrusive Runtime Verification

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 RV implies the instrumentation of system components. 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.

The usage of reconfigurable logic supporting versatile platform designs (e.g., soft-processors) enables innovative approaches to RV [10]. In particular, in the context of TSP systems, a design for TSP-oriented observers was proposed [11]. The *AIR Observer* (AO) features: *non-intrusiveness*, meaning system operation is not adversely affected; *flexibility*, meaning code instrumentation with RV probes is not required, although it may be used; *configurability*, being able to accommodate a set of different system-level, application-related and even mission-specific event observations.

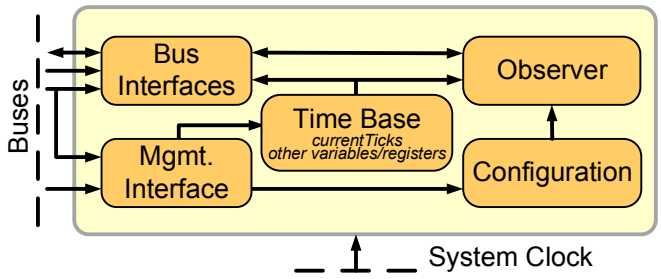

Fig. 4. AIR Observer architecture

The AO is plugged to the platform where the AIR software components execute, and comprises the hardware modules depicted in Figure 4: Bus Interfaces, capturing all physical bus activity, such as bus transfers or interrupts; Management Interface, enabling AO configuration; Configuration, storing the patterns of the events to be detected; Observer, detecting events of interest based on the registered configurations.

A **robust time base**[1] accounts for, in the AO hardware (Figure 4), the number of system-level clock ticks elapsed so far, to which AIR components have access, through the read only $currentTicks$ variable/register. For optimization purposes, other relevant read/write variables/registers may be available from the AO.

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

The AO continuously monitors the timeliness of AIR components and applications, functionally assuming a dual role: it detects when a given temporal bound is reached and/or if a given deadline was violated; it signals that it is time to perform a given (check) action, in order to verify/enforce timeliness.

IV. Mission's Timeliness Adaptation

The adaptation to changing environmental or operating conditions is crucial for unmanned space and aerial missions survivability, which can be significantly improved through software reconfigurability, as reported in [5].

The design of AIR integrates special-purpose mechanisms to address specific adaptation requirements, as thoroughly described in [6]. Aiming to improve its time domain behaviour, the *mode-based schedules* mechanism is reviewed.

A. Mode-based schedules

The original ARINC 653 notion of a single fixed PST [1], defined offline, hinder adaptation to changes in application requirements, according to the mission's phase, given certain functions may be required to execute only during some phases.

To address this primary limitation, AIR uses the notion of *mode-based partition schedules* [4], [6], inspired by the optional service defined within the scope of ARINC 653 Part 2 specification [12].

B. AIR mode-based schedules: original design and limitations

Instead of using one fixed PST, AIR-based systems can be configured with multiple PSTs, which may differ in terms of the MTF duration, of which partitions are scheduled, and of how much processor time is assigned to them, as shown in Figure 2. The system can switch between different PSTs; selection of the active PST is performed through a service call issued by an authorized and/or dedicated partition.

In the original definition of AIR *mode-based schedules*, a PST switch request is only effectively granted at the end of the ongoing MTF. This simple approach ensures that every process in all partitions have executed completely, upon a PST switch. Thus: applications are in a coherent state; PST switching is in conformity with the specified application timing. This model is adequate for long-term stable adaptation, such as entering a different mission's phase.

C. Redesigning AIR mode-based scheduling

During the ongoing MTF, a response to sudden and unexpected events (such as, a warning of an imminent collision) may be adversely delayed by the execution of functions, defined in the active PST, which do not have the capability of reacting to those (critical) events.

To extend the number/duration of periods where the executing functions have the capability of responding to critical events, a different schedule is required. The selection/activation of a new schedule is enhanced in two ways:

- by design, the new schedule is activated as soon as no critical activity is executing;
- by PST definition and configuration, the new schedule assigns execution time windows only to critical activities.

The first condition implies that, each partition needs to be classified as having its execution as critical or non-critical and that the time boundaries delimiting the execution of the critical execution periods need to be registered in the AO, both for monitoring purposes and to avoid ill-timed mode changes.

Secondly, for each mission phase, three schedules should be provided, each corresponding to a mode (see Figure 5), as follows:

- **normal** - corresponding to the normal execution of the activities defined for the mission;
- **survival** - meaning some severe external/internal condition that puts the vehicle and/or the mission in risk has been detected. This state is entered in response to the issuing of a SET_MODE_SCHEDULE primitive (Table I), by some system/application event handler (Figure 3). The schedule for this phase/mode shall allocate processor time only to fundamental avionic functions, in order to ensure safe and secure operation.
- **recovery** - the operation of the vehicle is no longer in risk, as confirmed, at all levels, by the RV mechanisms. A relevant system/application component issues a further SET_MODE_SCHEDULE primitive. Processing could now include full FDIR activities that, once accomplished, may allow the return to the normal mode.

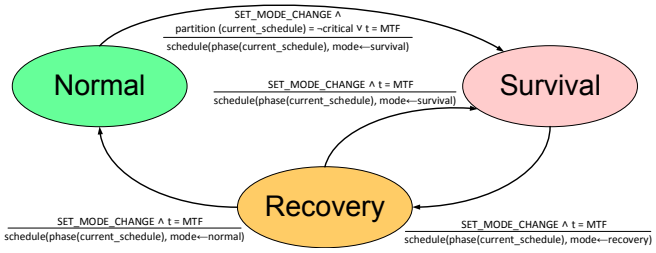

Fig. 5. Function schedule modes and allowed transitions

Hosting multiple PSTs aboard autonomous vehicles opens room for the (self-)adaptability of unmanned missions, in function of passage of time and of changing environmental and operational conditions. The use of full-fledged mode-based schedules contributes to a timely response to sudden changes in the operational conditions.

Pre-generation of different partition schedules can be aided by a tool that applies rules and formulas to the temporal requirements of processes/partitions, taking into account the functions' needs in different anticipated conditions [4], [13]. Unforeseeable conditions can be handled thorough the mechanisms for remote update of onboard software and PSTs [8].

V. Implementing Mission's Timeliness Adaptation

The proposed integration of non-intrusive RV and timeliness adaptation features follows the hardware-assisted approach described in [11]. This hardware/software co-design allows to maintain some degree of AIR architectural flexibility with advantages in terms of improved safety and timeliness, being specially interesting for running AIR in platforms integrating processor cores (e.g., dual-core ARM) and FPGA logic [14].

Algorithm 1 AIR Partition Scheduler with runtime verification featuring adaptation through mode-based schedules

1: ▷ Entered upon exception: partition preemption point signalled by the AO
2: ▷ Runtime verification actions
3: **if** $(mode(currentSchedule) = mode(nextSchedule) \ \wedge$
$schedules_{currentSchedule}.table_{tableIterator}.tick \neq$
$(currentTicks - lastScheduleSwitch) \bmod$
$schedules_{currentSchedule}.mtf) \vee$
$(mode(currentSchedule) \neq mode(nextSchedule) \ \wedge$
$schedules_{currentSchedule}.table_{tableIterator}.critical >$
$(currentTicks - lastScheduleSwitch) \bmod$
$schedules_{currentSchedule}.mtf)$ **then**
4: HealthMonitor($activePartition$)
5: **else** ▷ Partition Scheduling Table (PST) and partition switch actions
6: **if** $currentSchedule \neq nextSchedule \ \wedge$
$((mode(currentSchedule) \neq mode(nextSchedule) \ \wedge$
$(currentTicks - lastScheduleSwitch) \bmod$
$schedules_{currentSchedule}.mtf \geq$
$schedules_{currentSchedule}.table_{tableIterator}.critical) \ \vee$
$((currentTicks - lastScheduleSwitch) \bmod$
$schedules_{currentSchedule}.mtf = 0))$ **then**
7: ▷ PST switch actions
8: $currentSchedule \leftarrow nextSchedule$
9: $lastScheduleSwitch \leftarrow currentTicks$
10: $tableIterator \leftarrow 0$
11: **end if**
12: ▷ Partition switch actions
13: $heirPartition \leftarrow$
$schedules_{currentSchedule}.table_{tableIterator}.partition$
14: $tableIterator \leftarrow (tableIterator + 1) \bmod$
$schedules_{currentSchedule}.numberPartitionPreemptionPoints$
15: **end if**

A. AIR full-fledged mode-based scheduling

In a hardware-assisted approach to the implementation of AIR full-fledged mode-based scheduling, partition scheduling switch decisions from the AO hardware are complemented with software RV and partition switch actions: when a partition is dispatched, the absolute value (in POS-level clock ticks) of its *partition preemption point* is inserted in the AO configuration; when this instant is reached, an AO's hardware exception triggers the execution of Algorithm 1.

In **Algorithm 1**, if no mode change is claimed, the RV actions check (line 3), from the active PST, if the current instant is a partition preemption point. If a mode change is pending, the RV actions ensure (line 3) that no critical activity is executing at this instant. If none of these conditions apply, a severe system level error has occurred and the HM is notified (line 4) to handle the situation. Otherwise, the conditions for a **full-fledged mode-based** partition switch are checked in line 6: a PST schedule switch request is pending; a mode change switch is claimed and no critical activities are executing at the current instant or the current instant is the end of the MTF. If these conditions apply, the PST switching actions specified in [4], [11] are applied (lines 7-10) and a different PST will be used henceforth (line 8). The remaining lines of Algorithm 1 (lines 13-14) implement the conventional partition switch actions of [4], [11]. The processing resources to be assigned to the heir partition, until the next partition preemption point, are obtained from the PST in use (line 13). The AIR Partition Scheduler is set (line 14) to access the heir partition parameters.

Algorithm 2 AIR Partition Dispatcher

1: ▷ Entered from the AIR Partition Scheduler after partition switch actions
2: SAVECONTEXT($activePartition.context$)
3: $activePartition.lastTick \leftarrow currentTicks - 1$
4: $elapsedTicks \leftarrow currentTicks - heirPartition.lastTick$
5: $activePartition \leftarrow heirPartition$
6: REPLACEPREEMPTIONPOINT($heirPartition.tick$)
7: REPLACECRITICALPOINT($heirPartition.critical$)
8: RESTORECONTEXT($heirPartition.context$)
9: PENDINGSCHEDULECHANGEACTION($heirPartition$)

B. AIR hardware-assisted partition dispatching

The partition switch actions are followed by the execution of the AIR Partition Dispatcher specified in **Algorithm 2**. The hardware-assisted optimizations of [11] are maintained with respect to the software-based approach [4]: suppression of specific elapsed clock ticks setting, which are not required because the partition dispatcher is always invoked after a partition switch; insertion of the next partition preemption point in the AO configuration (line 6). However, to allow the runtime verification of mode change requests, the value of the next time critical schedule bound is now also inserted in the AO. The remaining actions in Algorithm 2 are related to saving and restoring the execution context (lines 2 and 8) and evaluation of the elapsed clock ticks (line 4). Line 9 enforces the execution of pending actions the first time the partition is executed after a PST change [4]. This last point is specially sensitive, since abrupt mode changes may leave some partitions in an inconsistent state.

C. Extending the APEX interface

The implementation of AIR full-fledged mode-based scheduling implies the addition of new primitives to the APEX interface, summarized in Table I. The AIR PAL component provides the adequate encapsulation with respect to the registering of the schedule timing information in the AO. The primitives listed in Table I can only be issued from an authorized and/or dedicated partition.

TABLE I
EXTENDING APEX PRIMITIVES TO SUPPORT FULL-FLEDGED MODE-BASED SCHEDULES

Primitive	Short description
Need to register/update critical execution period bounds in the AO	
SET_MODE_SCHEDULE	Requests a mode change for a new schedule Served if/when no critical activities
SET_PHASE_SCHEDULE	Requests a new mission phase schedule Served in normal mode, at the end of a MTF
No need to register/update critical execution period bounds in the AO	
GET_MODE_SCHEDULE_ID	Obtains the current schedule identifier
GET_MODE_SCHEDULE_STATUS	Obtains the current schedule status

Although semantically different APEX primitives are listed in Table I for the long-term adaptation of mission phases and for a (fast) short-term (self-)adaptation, through mode changes, both primitives share the same method (i.e., the activation of a new schedule), thus being optimal with respect to fitting the previous design and implementation of AIR components.

D. Analysis and discussion

Critical software, namely that developed to go aboard an aerial or space vehicle, goes through a strict process of verification, validation and certification. Code complexity affects the effort required for that process.

The AIR hardware-assisted approach translates to a significant reduction of AIR software code complexity. Most AIR software-based components have constant time complexity, $\mathcal{O}(1)$: accesses to multielement structures are made by index, being independent of the number and position of the elements. Nevertheless, some components exhibit a linear time complexity. That is the case associated with the *Pending Schedule Change Actions* procedure (Algorithm 2 - line 9), which in the worst case wields $\mathcal{O}(n)$, being n the number of processes in the partition.

Similar considerations apply to timing issues. However, due to the highly effective (i.e., $\mathcal{O}(1)$) implementation of the AIR software-based approach, the analysis in [11] for the *AIR Partition Scheduler* and *AIR Partition Dispatcher* components has shown only a moderate improvement in time overheads.

The expected reduction in the *mode change response delay*, with the corresponding increase in the ability of AIR-based systems to timely respond to sudden and unexpected changes in operational conditions, is heavily dependent of the structure of the active PST.

The exact value of the normalized *mode change response delay*, $\mathcal{T}_{mcd}$, in general depends on the instant, t, a *mode change primitive* is issued. That value is given by:

$$\mathcal{T}_{mcd}\left(\mathcal{T}\right) = \sum_{i=1}^{ncp} \left(H(\mathcal{T} - \mathcal{T}_{cs,i}) - H(\mathcal{T} - \mathcal{T}_{ce,i})\right) \times \left(\mathcal{T}_{ce,i} - \mathcal{T}_{cs,i}\right) \quad (1)$$

where:

$$\mathcal{T} = mod\left(\frac{t}{\mathcal{T}_{MTF}}\right) \quad (2)$$

is the current instant t normalized with the major time frame duration, $\mathcal{T}_{MTF}$, through the module function, $mod()$. $H()$ is the Heaviside function, defined as:

$$H(\mathcal{T} - \mathcal{T}_0) = \begin{cases} 0 & \mathcal{T} < \mathcal{T}_0 \\ 1 & \mathcal{T} \geq \mathcal{T}_0 \end{cases} \quad (3)$$

Furthermore, the following parameters are defined:

- ncp - the number of periods executing critical activities;
- $\mathcal{T}_{cs,i}$ - the instant, normalized by $\mathcal{T}_{MTF}$, where the execution of the critical period i starts;
- $\mathcal{T}_{ce,i}$ - the instant, normalized by $\mathcal{T}_{MTF}$, where the execution of the critical period i ends.

The results obtained for the *mode change response delay* with different types of schedules is illustrated in Figure 6. In the first case, only critical activities are scheduled for execution and therefore the mode change can only be performed by the end of the MTF. The second schedule includes an initial period of critical activities followed by a (small) period

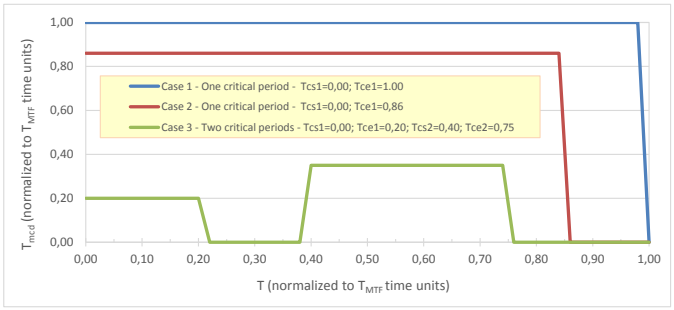

Fig. 6. Analysis of the *mode change response delay* obtained under different schedule scenarios

where mode change is allowed without delay. In the third and last schedule, illustrated in Figure 6, critical and non-critical activities are intermixed along the MTF, resulting in an overall decrease in the duration of the maximum and average periods where it is necessary to wait for a mode change to occur.

VI. Related Work

Reconfiguration and adaptation approaches have been applied in the realm of TSP systems and tested in avionic demonstrators [15]. Furthermore, non-instrusive runtime monitoring has been applied in embedded systems [16], [17] and, more specifically, in safety critical environments [18]. Configurable non-intrusive event-based frameworks for runtime monitoring have been developed within the embeddedd systems' scope [19], employing a minimally intrusive method for dynamic monitoring. Additionally, the RV concept has been applied to autonomous systems [20] and to a AUTOSAR-like RTOS, aiming the automotive domain [21]. [22] describes a runtime monitoring approach for autonomous vehicle systems requiring no code instrumentation by observing the network state. A unified framework for the specification, analysis and description of mode-change semantics applicable to real-time systems is presented in [23]. However, to the extent of our knowledge, no such techniques have been applied to TSP systems, specially if targeting avionic applications.

VII. Conclusion

This paper addressed fundamental mechanisms providing support for adaptive and self-adaptive behaviour to applications based on the AIR architecture for time- and space-partitioned systems. The usage of hybrid platforms combining processor cores and programmable logic makes advantageous the use of a hardware-assisted design complemented with some simple software-based components.

The introduction of full-fledged mode-base scheduling contributes for achieving a timely response to sudden and/or unexpected environmental and internal conditions, and enables improvements in both safety and timeliness properties. These mechanisms benefit from the use of non-intrusive runtime verification.

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. Additional work aims to take full advantage of multicore platforms in AIR, which include adaptation/reconfiguration features and, in the near future, extended RV capabilities.

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," ESA, Technical Note TEC-SW/09-247/JW, Aug. 2009.

[3] J. Rushby, "Partitioning in avionics architectures: Requirements, mechanisms and assurance," SRI International, California, USA, Tech. Rep. NASA CR-1999-209347, Jun. 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*, ser. LNCS, A. Casimiro, R. de Lemos, and C. Gacek, Eds., vol. 6420. Springer, 2010.

[5] M. Tafazoli, "A study of on-orbit spacecraft failures," *Acta Astronautica*, vol. 64, no. 2-3, pp. 195–205, 2009.

[6] 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.

[7] S. Santos, J. Rufino, T. Schoofs, C. Tatibana, and J. Windsor, "A portable ARINC 653 standard interface," in *Proc. 27th Digital Avionics Systems Conference*, 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 *Proceedings 9th IEEE Int. Conf. on Industrial Informatics (INDIN 2011)*, Caparica, Portugal, Jul. 2011.

[9] J. Carraca, R. C. Pinto, J. P. Craveiro, and J. Rufino, "Information security in time- and space-partitioned architectures for aerospace systems," in *Atas 6th Simpósio de Informática (INForum 2014)*, Porto, Portugal, Sep. 2014, pp. 457–472.

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

[11] J. Rufino, "Towards integration of adaptability and non-intrusive runtime verification in avionic systems," *SIGBED Review*, vol. 13, no. 1, Jan. 2016, (Special Issue on 5th Embedded Operating Systems Workshop).

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

[13] J. P. Craveiro and J. Rufino, "Schedulability analysis in partitioned systems for aerospace avionics," in *Proceedings 15th IEEE International Conference on Emerging Technologies and Factory Automation*, Bilbao, Spain, Sep. 2010.

[14] *ZYBO Reference Manual*, DILIGENT, Feb. 2014.

[15] G. Durrieu, G. Fohler, G. Gala, S. Girbal, D. G. Pérez, E. Noulard, C. Pagetti, and S. Pérez, "DREAMS about reconfiguration and adaptation in avionics," in *Proc. 8th Congress on Embedded and Real-Time Software and Systems (ERTS2016)*, Toulouse, France, Jan. 2016.

[16] C. Watterson and D. Heffernan, "Runtime verification and monitoring of embedded systems," *Software, IET*, vol. 1, no. 5, Oct. 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," Ph.D. dissertation, Carnegie Mellon University, USA, Feb. 2015.

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

[20] G. Callow, G. Watson, and R. Kalawsky, "System modelling for runtime verification and validation of autonomous systems," in *Proc. 5th Int. Conference on System of Systems Engineering*, Loughborough, UK, Jun. 2010, pp. 1–7.

[21] S. Cotard, S. Faucou, J.-L. Bechennec, A. Queudet, and Y. Trinquet, "A data flow monitoring service based on runtime verification for AUTOSAR," in *Proceedings of the 14th Int. Conf. on High Performance Computing and Communications*. Liverpol, UK: IEEE, Jun. 2012.

[22] A. Kane, O. Chowdhury, A. Datta, and P. Koopman, "A case study on runtime monitoring of an autonomous research vehicle (ARV) system," in *Proc. 15th Int. Conf. on Runtime Verification*, Vienna, Austria, Sep. 2015, pp. 102–117.

[23] L. T. X. Phan, I. Lee, and O. Sokolsky, "A semantic framework for mode change protocols," in *Proceedings of the 17th Real-Time and Embedded Technology and Applications Symposium*. IEEE, Apr. 2011.