Safe Online Reconfiguration of Time- and Space-Partitioned Systems

Joaquim Rosa, João Craveiro, and José Rufino
University of Lisbon, Faculty of Sciences, LaSIGE
FCUL, Ed. C6, Piso 3, Campo Grande, 1749-016 Lisbon, Portugal
{jrosa, jcraveiro}@lasige.di.fc.ul.pt, ruf@di.fc.ul.pt

***Abstract*—Future space missions call for advanced computing system architectures fulfilling strict size, weight and power consumption (SWaP) requisites, decreasing the mission cost and ensuring the safety and timeliness of the system. The AIR (ARINC 653 in Space Real-Time Operating System) architecture defines a partitioned environment for the development and execution of aerospace applications, following the notion of time and space partitioning (TSP), preserving application timing and safety requisites. Due to the change of the mission plans or in the presence of unexpected events, it may be necessary or useful to be able to reconfigure the scheduling of the system applications at execution time. In this paper we present an algorithm for updating application schedules, showing results from the proof-of-concept prototype in the scope of the AIR architecture.**

I. Introduction

Future space missions demand for innovative computing architectures and onboard software systems, meeting strict requisites of size, weight and power consumption (SWaP), thus decreasing the overall cost of the mission and obeying to strict safety and timeliness requirements.

Partitioned architectures implementing the logical separation of applications in criticality domains, named partitions, permit to host several partitions in the same computational infrastructure, thus fulfilling the SWaP requirements [1]. The notion of temporal and spatial partitioning (TSP) implies that the execution of applications in one partition does not affect other partitions' timing requisites and that separate addressing spaces are assigned to different partitions [2].

The design of AIR Technology has been supported by the interest of the space industry partners, especially the European Space Agency (ESA), in applying the TSP concepts to the aerospace domain [3]. A typical spacecraft hosts several subsystems consisting of avionics functions and payload, which closely interact with each other. Relevant examples are the Attitude and Orbit Control Subsystem (AOCS), Communications, Onboard Data Handling (OBDH), and Telemetry, Tracking and Command (TTC). In TSP systems, the several functions share the same computational resources being hosted in different partitions.

This work was partially developed within the scope of the European Space Agency Innovation Triangle Initiative program, through ESTEC Contract 21217/07/NL/CB, Project AIR-II (ARINC 653 in Space RTOS – Industrial Initiative, http://air.di.fc.ul.pt). This work was partially supported by Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology), through the Multiannual Funding and CMU-Portugal Programs and the Individual Doctoral Grant SFRH/BD/60193/2009.

Many episodes in the history of space missions showed that, during the course of a mission, it may be useful, necessary or even primordial to introduce new functions in spacecraft or modify existing ones to deal with unexpected external events and internal failures. The potential to adapt to changing environmental or operating conditions is of great importance for a mission's survival. NASA's Mars Pathfinder is an example of a mission where the possibility to remotely modify the system's configuration was crucial for its survival [4]. Flexible adaptation to unforeseen events has been proven to prolong the lifetime of space vehicles by years [5].

This paper addresses reconfiguration in time- and space-partitioned systems enabling onboard update of partition scheduling tables (PSTs). The methodology defined here was motivated by spaceborne systems' need to adapt to changing conditions and unexpected events. The remainder of this paper is organized as follows. Section II briefly describes the AIR architecture. AIR reconfigurability and adaptability features are addressed in Section III. The specific components added to secure safe PST reconfiguration are explained in Section IV. Section V analyses the complexity for the update of PSTs. Section VI presents the proof-of-concept prototype and relevant results. Section VII describes the related work. Finally, Section VIII issues concluding remarks and future research directions.

II. AIR Technology

The AIR Technology is currently evolving to the definition of an industrial product through the improvement and completion of the architecture design and engineering process.

A. System architecture

The AIR architecture, illustrated in Fig. 1, relies on the *AIR Partition Management Kernel* (PMK) to enforce robust TSP. A (real-time or generic) operating system herein referred as *Partition Operating System* (POS), is provided per partition. Each POS is wrapped by the *AIR POS Adaptation Layer* (PAL) hiding its particularities from other AIR components [3].

An AIR-based system provides a way to achieve the containment of faults to the domain where they occur using the architectural principle of robust TSP. Temporal partitioning ensures that the real-time requisites of the different functions executing in each partition, as ensured by the partition's scheduling policy, are not affected by the coexistence with

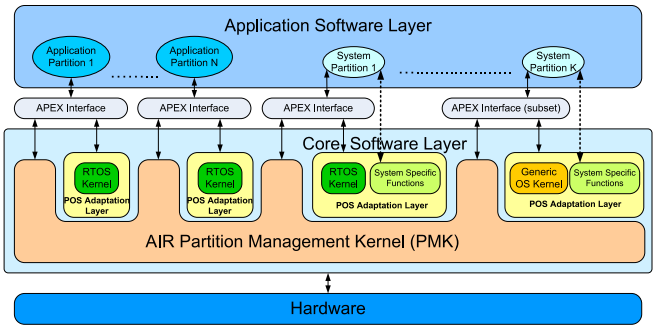

Fig. 1. AIR system architecture and integration of partition operating systems

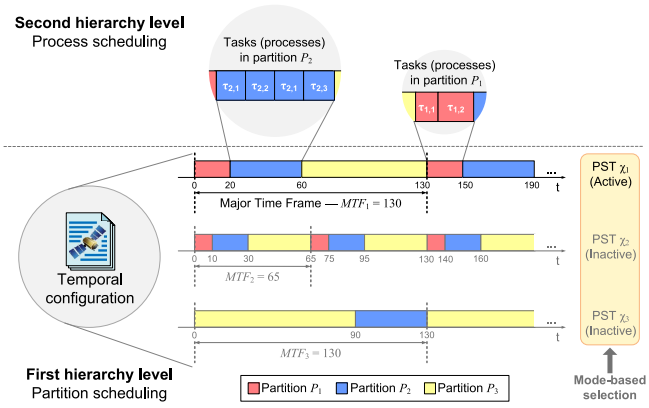

Fig. 2. Two-level mode-based partition scheduling

other functions in other partitions. The spatial partitioning relies on having dedicated addressing spaces for applications executing on different partitions.

The *Application Executive* (APEX) *interface* component provides a standard programming interface with a service definition derived from the ARINC 653 specification [6]. The set of available services concerns partition and process management, time management, intra and interpartition communication and health monitoring. The AIR architecture implements the advanced notion of *Portable APEX*, ensuring portability between the different POSs [7]. The original APEX interface has been extended to cope with services related to onboard update operations [8].

B. Scheduling Partitions

The AIR architecture uses a two-level scheduling scheme, where partitions are scheduled under a predetermined sequence of time windows, cyclically repeated over a *major time frame* (MTF). In each partition, the respective processes are scheduled according to the native operating system's process scheduler (Fig. 2).

The AIR Technology design incorporates the notion of *mode-based partition schedules* to address timeliness and fault tolerance limitations in the original ARINC 653 notion of a single fixed PST, defined offline [6]. The support for mode-based schedules requires the provision of additional APEX services. The APEX_SWITCH_SCHEDULE service, represented in Algorithm 1 sets the schedule that will start executing at the begin of the next MTF. This primitive receives, as its only parameter, the $scheduleId$, referencing the index of the next PST to be used.

The *AIR Partition Scheduler* is responsible for guaranteeing to make a schedule switch effective at the end of the respective MTF and functions as described in the Algorithm 2. The first verification to be made is whether the current instant is a partition preemption point (Algorithm 2, line 2). In case it is not, the execution of the partition scheduler is over; this is both the best case and the most frequent one. If it is a partition preemption point, a verification is made (Algorithm 2, line 3) as to whether there is a pending schedule switch to be applied and the current instant is the end of the MTF. A pending schedule switch is originated by a request to change to a different PST (Algorithm 1). Since a schedule switch

Algorithm 1 APEX_SWITCH_SCHEDULE primitive

```
1: function APEX_SWITCH_SCHEDULE(scheduleId)
2:     nextSchedule ← scheduleId
3: end function
```

Algorithm 2 AIR Partition Scheduler (mode-based schedules)

```
1: ticks ← ticks + 1    ▷ ticks: global system clock tick counter
2: if schedules_currentSchedule.table_tableIterator.tick =
     (ticks − lastScheduleSwitch) mod
     schedules_currentSchedule.mtf then
3:     if currentSchedule ≠ nextSchedule ∧
         (ticks − lastScheduleSwitch) mod
         schedules_currentSchedule.mtf = 0 then
4:         currentSchedule ← nextSchedule
5:         lastScheduleSwitch ← ticks
6:         tableIterator ← 0
7:     end if
8:     heirPartition ←
         schedules_currentSchedule.table_tableIterator.partition
9:     tableIterator ← (tableIterator + 1) mod
         schedules_currentSchedule.numberPartitionWindows
10: end if
```

happens only at the end of the current MTF, in order to maintain the timeliness, this may result on a waiting time before the PSTs switching [3]. If the referred conditions apply, then a different PST will be used henceforth (Algorithm 2, line 4). The partition which will hold the processing resources until the next preemption point, dubbed the heir partition, is obtained from the PST in use (Algorithm 2, line 8) and the AIR Partition Scheduler will now be set to expect the next partition preemption point (Algorithm 2, line 9) [3].

III. Reconfigurability and Adaptability

Adaptation to changing or unexpected conditions is of great importance for a mission's survival. The AIR architecture employs several adaptability mechanisms, to maintain and improve the system effectiveness when facing internal or external changes. This enables the safe reconfiguration of system components. Reconfigurability may also involve self-adaptability, which concerns the system's autonomous interpretation and adaptation to environmental changes.

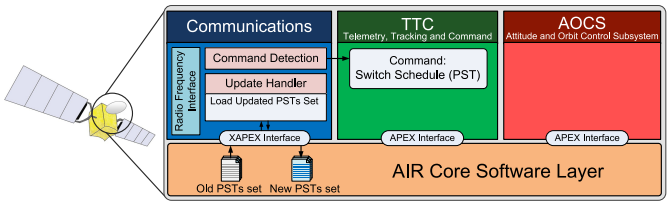

Fig. 3. Spacecraft time- and space-partitioned computing platform

A. Achieving reconfigurability

The support for robust reconfiguration in the AIR architecture is done through mechanisms such as mode-based scheduling; health monitor, responsible for handling and containing errors to their domains of occurrence; process deadline violation monitoring, to detect violations of the processes' timing constraints, and; low-level event overload control, to control the timeliness of asynchronous events [9], [3].

B. Achieving (self-)adaptability

By offering the possibility to host natively multiple PSTs and switch among them on demand during the execution of the system, AIR permits the (self-)adaptation of the system to the mission's different phases and to operational condition changes [9]. For example, a request to use a different set of PSTs can be issued by the ground mission control or autonomously by the onboard system through the spacecraft Attitude and Orbit Control Subsystem (AOCS), when an event implies changing the partitions' temporal requirements.

IV. Safe Update of Partition Schedules

We define a methodology to permit the update of partition scheduling tables on spaceborne TSP systems during a mission. The challenges faced are related to maintaining the real-time and safety guarantees of the original mission [8]. This should not affect the correct overall behaviour of the system, including the timeliness of the already running applications.

A. Integration on spacecraft onboard platform

The safe update of PSTs is carried out by the *Update Handler*, which is a process/thread integrated in the spacecraft's partition hosting the Communications functions. The Update Handler, illustrated in Fig. 3, is responsible for handling and managing the updates of PSTs. This involves providing the received sets of PSTs to the Partition Scheduler component of the AIR PMK. The update operation is supported by a (secure) communication channel and data communication protocol. The Communications partition also includes a component for command detection, which passes the commands issued from the ground mission control to the TTC through an interpartition communication channel. The example illustrated in Fig. 3 includes a ground command to switch schedule.

B. The XAPEX_PSTUPDATE service

To support the introduction of the operation for updating PSTs, the original APEX interface was extended with an appropriate service, referred as XAPEX_PSTUPDATE. This primitive, provided by an extended APEX (XAPEX) interface, is available only to specifically authorized partitions, such as the one responsible for the spacecraft communications, as illustrated in Fig. 3.

Algorithm 3 XAPEX_PSTUPDATE primitive

```
1: function XAPEX_PSTUPDATE(newSchedules)
2:     while ¬safePstUpdate do
3:         if currentSchedule = nextSchedule then
4:             for newSchedules_i ∈ newSchedules do
5:                 if newSchedules_i ≡ schedules_currentSchedule then
6:                     safePstUpdate ← TRUE
7:                     newCurrentSchedule ← i
8:                     newNextSchedule ← i
9:                     break
10:                end if
11:            end for
12:        end if
13:    end while
14:    SWAP(schedules,newSchedules)
15:    currentSchedule ← newCurrentSchedule
16:    nextSchedule ← newNextSchedule
17: end function
```

C. Methodology for onboard update of PSTs

The complete onboard update methodology consists of the definition of a new set of PSTs and the modification of systemwide configurations, to upgrade the original mission and reconfigure it according to new requirements. The activation of the new set of PSTs must guarantee the safety of switching between the old and the new sets of PSTs (see Fig. 4), thus ensuring that the correctness of system scheduling is not compromised. The update methodology consists of a four-step procedure described as follows:

1) Offline verification and validation of redefined PSTs: The definition of new sets of PSTs must involve the verification and validation of such components. This aims to secure the correctness of the redefined set of PSTs thus ensuring that the safety and timeliness of the target system would not be compromised [10].

2) Formatting of redefined PSTs: The general goal of this step is to create an object file consisting of the new set of PSTs. This object file should be built according to a specific format in order to be recognized by the Update Handler. After this step, the object file with the new set of PSTs will be uploaded to the spacecraft onboard computer.

3) Transfer of redefined PSTs: In the spacecraft, the PSTs are received by the partition hosting the communication functions (Fig. 3). The Update Handler inspects the uploaded object, recognizes it as a set of PSTs and invokes the XAPEX_PSTUPDATE primitive to issue a request to apply the set of PSTs updated.

4) Activation of redefined PSTs: The first condition to the safe application of a new set of PSTs is that a schedule switch is not pending (Algorithm 3, line 3). This further means that a request to switch to another schedule was not issued, through the invocation of the APEX_SWITCH_SCHEDULE primitive (Algorithm 1). The activation of the new set of PSTs will

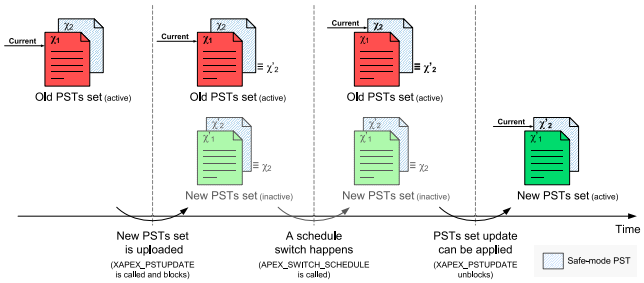

Fig. 4. Update of a set of PSTs

only become effective at the end of the current MTF. This condition prevents an eventual switch to an unexpected PST. The second condition to the safe application of a new set of PSTs is that the currently selected schedule has an identical counterpart in the new PSTs set (Algorithm 3, lines 4–8). In other words, a requested PSTs set update operation can be performed if the set of the modified PSTs received has a non-empty subset identical to a subset of the PSTs set currently active on the system. This avoids a non-requested schedule switch after applying the update. If this second condition is not met, the update will only be applied when a switch to a PST which meets the said criterion occurs. This scenario is illustrated in Fig. 4. To provide the possibility to update currently operational PSTs, a safe-mode PST (which is guaranteed to always exist on both the old and the new PSTs set) can be employed (represented in Fig. 4 as χ_2 and χ'_2).

V. PSTs Update Algorithm Analysis

The requirements for code efficiency and bounded execution times should be met during the implementation of the update of the PSTs set operation, although maintaining the safety and timeliness of the remaining system functions.

A. Code complexity

Code complexity increases the probability of there being software bugs and requires more efforts on the verification, validation and certification process. A metric for code complexity concerns its size, in source lines of code (SLOC). To compare programs written by distinct developers, the use of standardized accounting methods is required, such as the logical source lines of code (logical SLOC) metric of the Unified CodeCount tool [11]. Other typical software metric is the cyclomatic complexity (CC), which gives an upper bound for the number of tests needed for full branch coverage, and a lower bound for those needed for full path coverage. The Table I shows the logical SLOC and CC values for the C implementation of the XAPEX_PSTUPDATE primitive and the AIR Partition Scheduler [9].

B. Computational complexity

We analyse the computational complexity of the XAPEX_PSTUPDATE primitive (see Algorithm 3). Access to multielement structures, such as $schedules$ and $newSchedules$, is made by index thus the inherent complexity does not depend on the number of elements.

TABLE I
Logical SLOC and cyclomatic complexity (CC) for the XAPEX_PSTUPDATE primitive and the AIR Partition Scheduler

	Logical SLOC	**CC**
XAPEX_PSTUPDATE	11	4
AIR Partition Scheduler	13	4

Searching the set of the updated PSTs ($newSchedules$ in Algorithm 3) to find one PST that matches with the currently selected PST ($schedules_{currentSchedule}$ in Algorithm 3) is a linear operation (lines 4 and 5). In the best case, this wields $\mathcal{O}(1)$, which happens if the first PST in the set of the updated PSTs is identical to the one currently selected. In the worst case, this operation wields $\mathcal{O}(n)$, where n is the number of PSTs, since we may need to compare all the PSTs in the updated set until we reach one that is identical to the PST currently active. Verifying if two PSTs are identical is also a linear operation (Algorithm 3, line 5). This comparison involves verifying whether the MTF values and the number of preemption points of the two PSTs being compared are equal, and; verifying, for each preemption point, whether the same clock tick corresponds to the same partition. PSTs that do not meet these conditions are considered non-identical. This operation wields $\mathcal{O}(m)$, where m is the number of preemption points ($numberPartitionWindows$ in Algorithm 2). The remaining XAPEX_PSTUPDATE instructions wields $\mathcal{O}(1)$.

The overall computational complexity of the XAPEX_PSTUPDATE primitive wields $\mathcal{O}(m) \times \mathcal{O}(n) = \mathcal{O}(mn)$. In practice, n corresponds to the number of different mission phases. The expected value for m is the maximum number of partition preemption points.

VI. Proof-of-concept Prototype and Evaluation

A. Prototyping

Aiming to demonstrate the onboard update of partition scheduling tables, we modified an existing prototype of an AIR-based system to include facilities for the update of PSTs. This prototype includes four partitions, each one running a RTEMS-based mockup application [12] representing typical spacecraft functions (Fig. 5). Partition P_1 is associated to the AOCS functions; P_2 features the Communications functions, being responsible for the execution of the Update Handler; P_3 concerns OBDH, and; P_4 features the TTC operations.

In order to allow the visualization and interaction during the proof of concept demonstration, the prototype takes profit of VITRAL, a text-mode windows manager for RTEMS [13], illustrated in Fig. 5. Each partition has its own output window, which presents relevant information concerning the partitions' applications. There are also two windows allowing the observation of the behaviour of AIR components. For demonstration purposes, the support for keyboard interaction allows the activation of the update of PSTs (Algorithm 3) and the switching between different partition scheduling tables (Algorithm 1). The demonstration was implemented for an Intel IA-32 target platform and tested on the QEMU emulator [14].

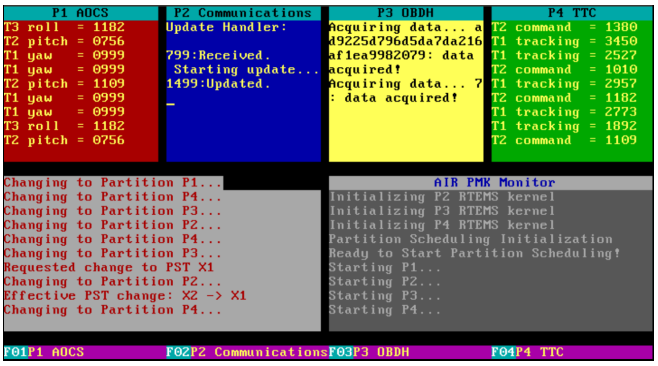

Fig. 5. Prototype implementation demonstration, featuring the VITRAL text-mode windows manager for RTEMS

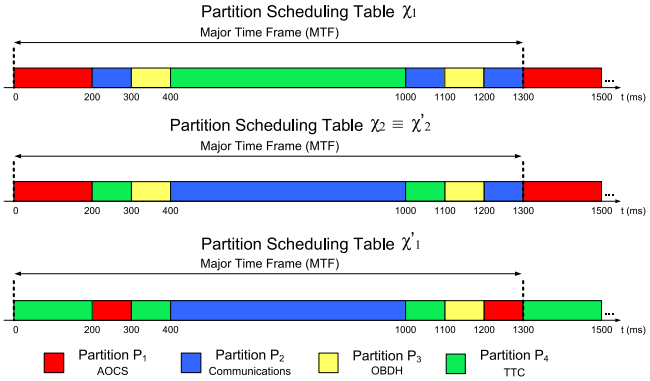

Fig. 6. Partition scheduling tables for the prototype implementation

B. Evaluation: test scenarios and results

We analyze the functional behaviour of the partition scheduling during the update of a new set of PSTs. In order to test different scenarios and compare the results, we set up the demonstration with different possible configurations. Then, we perform several operations in a specific execution order. These operations concern the activation of the PSTs set update, which is achieved through a call to the XAPEX_PSTUPDATE primitive (Algorithm 3), described in Section IV, and; the request to switch to a different schedule, through a call to the APEX_SWITCH_SCHEDULE primitive (Algorithm 1), described in Section II-B. In the real world, a request to change to a different schedule may be either issued autonomously by the spacecraft or upon decision from the ground control [3].

For demonstration purposes, the system is configured with a set of two PSTs, χ_1 and χ_2. The set of redefined PSTs is composed by two PSTs, χ'_1 and χ'_2, described in Fig. 6 and Table II. The PST χ'_1 is an update of χ_1, whereas the PST χ'_2 is identical to χ_2 and therefore both assume the role of safe-mode PSTs (Fig. 4). The referred PSTs have all a MTF of 1300 time units.

At first, we will define four base scenarios and then we discuss some variations. The following test scenarios 1 to 4 cover those four cases, which concern update currently active/inactive PSTs with/without a pending switch schedule request (Algorithm 3, line 3).

Test scenarios:

1) The initial PST is χ_1. There is no switch schedule request pending (Algorithm 3, line 3). The update of PSTs set is requested, simulating the issuing of a command with this purpose from the ground mission control, but the new set of PSTs is not activated.
   Result: The system continues its execution according to PST χ_1. Since a condition required to the safe update was not accomplished (Algorithm 3, line 5), the update cannot be applied. The XAPEX_PSTUPDATE algorithm invoked by partition P_2 remains in loop (Algorithm 3, line 2) until the referred condition is reached, which only occurs if a request to switch to PST χ_2 is issued. This scenario is addressed in Fig. 4.

TABLE II
PARTITION SCHEDULING TABLES USED IN TEST SCENARIOS

Preemption point (time units)	Partitions (per PST) χ_1	$\chi_2 \equiv \chi'_2$	χ'_1
0	P_1	P_1	P_4
200	P_2	P_4	P_1
300	P_3	P_3	P_4
400	P_4	P_2	P_2
1000	P_2	P_4	P_4
1100	P_3	P_3	P_3
1200	P_2	P_2	P_1

MTF = 1300 time units

2) The initial PST is χ_1. There is no switch schedule request pending (Algorithm 3, line 3). A switch to PST χ_2 is requested. Then, the update of PSTs set is activated. After the current MTF, the system starts being scheduled by χ_2. When the process which remained blocked on the XAPEX_PSTUPDATE primitive's loop is scheduled for execution, during a time window assigned to partition P_2, the update is applied (the conditions represented in lines 3 and 5 of the Algorithm 3 were accomplished). Following, a switch to PST χ_1 is requested.
   Result: After the end of the current MTF, the system starts being scheduled by the updated PST χ'_1.
3) The initial PST is χ_2. There is no switch schedule request pending (Algorithm 3, line 3). The update of PSTs set is activated. The update is applied during a time window of the partition P_2 (the conditions represented in lines 3 and 5 of the Algorithm 3 were accomplished). Following, a switch to PST χ_1 is requested.
   Result: After the end of the current MTF, the system starts being scheduled by the updated PST χ'_1.
4) The initial PST is χ_2. There is no switch schedule request pending (Algorithm 3, line 3). A switch to PST χ_1 is requested. Then, the update of PSTs set is activated.
   Result: After the end of the current MTF, the system starts being scheduled by the PST χ_1. The update is not applied (the condition represented in line 3 of the Algorithm 3 was not accomplished). This will happen eventually when the system conditions change. In this

case we face another scenario, namely the test scenario 2. The update would only be applied during the execution of the Update Handler process in the partition P_2, after an effective schedule switch to the PST χ_2. Thus, on the next schedule switch to PST χ_1, the system would start being scheduled by the updated PST χ_1'.

Additional tests concern modifying χ_1' and simulate the update of a new set of PSTs to the spacecraft. The purpose of these tests was to verify that the obtained results were in conformity with those achieved in the four base scenarios previously described. The first additional test concerns defining χ_1' with different time window durations. The second additional test defines a PST χ_1' with no time window attributed to partition P_3. Switching to this PST may be useful in a mission phase that requires more processing time to be assigned to a specific spacecraft function. For example, during an orbit insertion maneuver, the AOCS may require more processing time than the OBDH. The third additional test uses an MTF of 650 time units in the definition of χ_1'. The results obtained through the alternative definitions of the PST χ_1 correspond to the same as those described in the test scenarios 1 to 4.

VII. Related Work

There are different approaches regarding aerospace system reconfiguration. In airborne systems, these concern complex online or offline techniques to ensure the flight or mission's effectiveness. The method proposed in [15] uses multi-static reconfiguration, which consist of the activation of a predefined configuration, selected autonomously according to the system health state at system startup. Moreover, the in-field verification and validation of system configurations and software components have a remarkable importance to determine whether it is safe to proceed with update operations and reconfigurations [16], [17]. Online update methodologies, such as the one approached in this paper, may benefit from the use of techniques of dynamic software update, which refers to the modification of software components without stopping the system execution [18]. Different techniques of dynamic software update have been studied and developed, including in the domain of real-time systems [19], [20]. However, these do not directly apply to aerospace TSP systems.

VIII. Conclusion and Future Work

In this paper, we presented a methodology for the update of partition scheduling tables, addressing the requisites of time- and space-partitioned systems. The update of partition scheduling tables is motivated by the need to adapt to changing and unexpected environmental conditions, and to overcome severe incidents or internal failures during the system operation. We discussed and detailed the algorithm behind the onboard update methodology. The possibility to reconfigure system parameters, such as partition scheduling tables, is extremely important since it may contribute to reach a safe system upon the occurrence of environmental changes or spacecraft failures, thus increasing the mission's survivability.

Future development involves the update of application software components hosted in partitions. This must attend, at first, partitions in the idle mode, and next, partitions in the active mode. The latter may benefit from the use of dynamic software update techniques. The update of critical software components without interrupting the system execution is foreseen. Other future challenges fall in remote system monitoring and modification of systemwide control parameters.

References

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

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

[3] 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. Berlin Heidelberg: Springer, 2010.

[4] M. Jones, "What really happened on Mars Rover Pathfinder," The Risks Digest (http://catless.ncl.ac.uk/Risks), Forum on Risks to the Public in Computers and Related Systems Issue 49, Dez 1997.

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

[6] AEEC (Airlines Electronic Engineering Committee), "Avionics application software standard interface, part 1 - required services," Aeronautical Radio, Inc., ARINC Spec. 653P1-2, Mar. 2006.

[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. Craveiro, and J. Rufino, "Adaptability and survivability in spaceborne time- and space-partitioned systems," in *EUROCON 2011 - International Conference on Computer as a Tool*, Lisboa, Portugal, Apr. 2011.

[9] J. Craveiro and J. Rufino, "Adaptability support in time- and space-partitioned aerospace systems," in *Proceedings of the Second International Conference on Adaptive and Self-adaptive Systems and Applications (ADAPTIVE 2010)*, Lisboa, Portugal, Nov. 2010, pp. 152–157.

[10] J. Craveiro, J. Rufino, and F. Singhoff, "Architecture, mechanisms and scheduling analysis tool for multicore time- and space-partitioned systems," in *23rd Euromicro Conference on Real-Time Systems (ECRTS 2011) — Work-in-Progress session*, Porto, Portugal, Jul. 2011.

[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] *RTEMS C User's Guide*, 4th ed. On-Line Applications Research Corporation, 2010.

[13] M. Coutinho, C. Almeida, and J. Rufino, "VITRAL - a text mode window manager for real-time embedded kernels," in *Proc. of the ETFA 2006*, Prague, Czech Republic, Sep. 2006, pp. 1254–1260.

[14] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005.

[15] C. Engel, A. Roth, P. H. Schmitt, R. Coutinho, and T. Schoofs, "Enhanced dispatchability of aircrafts using multi-static configurations," in *Proceedings of the Embedded Real Time Software and Systems (ERTS^2 2010)*, Toulouse, France, 2010.

[16] M. Neukirchner, S. Stein, H. Schrom, and R. Ernst, "A software update service with self-protection capabilities," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, Dresden, Germany, Mar. 2010, pp. 903–908.

[17] A. T. Bahill and S. J. Henderson, "Requirements development, verification, and validation exhibited in famous failures," *Systems Engineering*, vol. 8, no. 1, pp. 1–14, 2005.

[18] M. Hicks, "Dynamic software updating," *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 6, Nov. 2005.

[19] J. Montgomery, "A model for updating real-time applications," *Real-Time Syst.*, vol. 27, no. 2, pp. 169–189, 2004.

[20] M. Wahler, S. Ritcher, and M. Oriol, "Dynamic software updates for real-time systems," in *Proc. HotSWUp'09*, Orlando, FL, USA, Oct. 2009.