UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

DEVELOPMENT AND UPDATE OF AEROSPACE APPLICATIONS IN PARTITIONED ARCHITECTURES

Joaquim Luís Santo Rodrigues Cleto Rosa

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

2011

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

DEVELOPMENT AND UPDATE OF AEROSPACE APPLICATIONS IN PARTITIONED ARCHITECTURES

Joaquim Luís Santo Rodrigues Cleto Rosa

DISSERTAÇÃO

Trabalho orientado pelo Prof. Doutor José Manuel de Sousa de Matos Rufino

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

2011

This work was partially funded by:

ESA/ITI - European Space Agency Innovation Triangular Initiative
(through ESTEC Contract 21217/07/NL/CB - Project AIR-II)

FCT - Fundação para a Ciência e a Tecnologia
(through the Multiannual Funding Programme)

Acknowledgements

This thesis represents for me the conclusion of five years of study with results which I could not achieve without the help and support of several people. I express herein my gratitude to all of them.

Firstly, I would like to thank my advisor, Prof. José Rufino, for his crucial help during the development of this thesis. His remarks and suggestions have always contributed to the increase of my knowledge and to develop further my work. His dedication and support while I was writing this thesis and related articles was truly important. His motivation and enthusiasm forced me to keep working even in the hardest moments, and to always choose my own way to achieve the proposed goals.

Secondly I would like to thank João Craveiro, Jeferson Souza, and Ricardo Pinto, for all the valuable insights given during my research work and academic route. Their availability, good advices, and friendship were extremely important for me.

Thirdly, I would like to thank Navigators and all of the LaSIGE fellows for the interesting and constructive discussions where I learnt a lot.

I also would like to thank all my colleagues from Informatics Engineering, especially those from 2006/2007 class, with whom I shared the same passion.

Moreover, I am very grateful to all my professors and people from the DI, for being so warm and helpful, and for having taught me so many interesting things. I thank FCUL for being my school and my home for the last five years.

I thank all my friends from FCUL, from whom I learnt interesting topics beyond Informatics, I thank new friends, for sharing their diversified life experiences, and to old friends, for always being there when I needed.

Lastly but not least, I would like to thank my family for all the support and opportunities given. Thank you for believing in me.

Lisboa, April 2011
Joaquim Luís Santo Rodrigues Cleto Rosa

À minha família e amigos.

Abstract

To face the challenges and requirements imposed by future space missions, the aerospace industry has been following the trend of adopting innovative and advanced computing system architectures fulfilling strict requisites of size, weight and power consumption (SWaP) thus decreasing the mission overall cost and ensuring the safety and timeliness of the system.

The AIR (ARINC 653 in Space Real-Time Operating System) architecture has been defined dependent on the interest of the aerospace industry, especially the European Space Agency (ESA). AIR provides a partitioned environment for the development and execution of aerospace applications, based on the idea of time and space partitioning (TSP), aiming the preservation of the application requirements, timing and safety.

During a space mission, the occurrence of unexpected events or the change of the mission plans introduces new constraints to the mission. Therefore, it is paramount to have the possibility to host new applications in spacecraft onboard computer platform, or modify the existing ones in execution time, thus fulfilling new requirements or enhancing spacecraft functions.

The work described on this thesis introduces in the AIR architecture the support for the inclusion of new features to the mission plan during the system operation. These new features may be composed of modified software components or the corresponding timing requirements. The improvement of the AIR architecture with the ability to perform software updates requires a suitable development environment and tools. Therefore, the methodology for software development in AIR-based systems, regarding the build and integration process, is reexamined.

Keywords: Aerospace systems and applications, software update, time and space partitioning (TSP), adaptability and reconfigurability, real-time embedded systems

Resumo

Para enfrentar os desafios e requisitos impostos por missões espaciais futuras, a indústria aeroespacial tem vindo a seguir uma tendência para adoptar arquitecturas computacionais inovadoras e avançadas, cumprindo requisitos estritos de tamanho, peso e consumo energético (SWaP) e assim diminuir o custo total da missão assegurando a segurança na operação e a pontualidade do sistema.

A arquitectura AIR (ARINC 653 in Space Real-Time Operating System), desenvolvida para responder ao interesse da indústria aeroespacial, particularmente da Agência Espacial Europeia (ESA), fornece um ambiente compartimentado para o desenvolvimento e execução de aplicações aeroespaciais, seguindo a noção de compartimentação temporal e espacial, preservando os requisitos temporais das aplicações e a segurança na operação.

Durante uma missão espacial, a ocorrência de eventos inesperados ou alterações aos planos da missão introduz novas restrições. Assim, é de grande importância ter a possibilidade de alojar novas aplicações na plataforma computacional de veículos espaciais ou modificar aplicações já existentes em tempo de execução e, deste modo, cumprir os novos requisitos ou melhorar as funções do veículo espacial.

O presente trabalho introduz na arquitectura AIR o suporte à inclusão e actualização de novas funcionalidades ao plano de missão durante o funcionamento do sistema. Estas funcionalidades podem ser formadas por componentes de software modificados ou pelos requisitos temporais correspondentes. O melhoramento da arquitectura AIR com a possibilidade de realizar actualizações de software requer um ambiente e ferramentas de desenvolvimento adequados. Neste sentido, a metodologia para o desenvolvimento de software em sistemas baseados na arquitectura AIR é revisitada.

Palavras-chave: Sistemas e aplicações aeroespaciais, actualização de software, compartimentação temporal e espacial, adaptabilidade e reconfigurabilidade, tempo-real

Resumo alargado [1]

Missões espaciais futuras precisam de uma nova geração de veículos espaciais. Esta necessidade tem suscitado o interesse de agências espaciais e parceiros industriais na definição e desenho dos pilares de construção fundamentais para novas plataformas computacionais a bordo de veículos espaciais, onde as necessidades estritas de confiabilidade, pontualidade, segurança na operação e segurança da informação são combinadas com um requisito global para reduzir o tamanho, peso e consumo energético da infraestrutura computacional.

A definição de arquitecturas compartimentadas que implementem a contenção lógica de aplicações em domínios de criticalidade, designados por partições, permite alojar diferentes partições na mesma infraestrutura computacional e possibilita o cumprimento dos requisitos referidos. A noção de compartimentação temporal e espacial assegura que as actividades de uma partição não afectam a pontualidade das actividades em execução noutras partições e impede as aplicações de acederem ao espaço de endereçamento de cada uma das outras.

No contexto das aplicações aeroespaciais suportadas por uma arquitectura compartimentada, as diversas funções de um veículo espacial, tais como o subsistema de Controlo Orbital e de Orientação, ou o subsistema de Telemetria, Rastreio e Comando, partilham os mesmos recursos computacionais e ao mesmo tempo mantêm a sua independência no que diz respeito à contenção de eventuais faltas e à validação e verificação de software, ficando alojadas em partições distintas. No domínio temporal, as diferentes partições são escalonadas de acordo com tabelas de escalonamento fixas e cíclicas.

Para a escrita, construção, depuração, e manutenção das aplicações em arquitecturas compartimentadas é necessário ultrapassar algumas limitações próprias das ferramentas clássicas de geração de software, que não foram desenhadas para o desenvolvimento deste género de aplicações. Identificadas as limitações, um dos desafios é encontrar soluções e adaptar as ferramentas já existentes de forma a conseguir ter um ambiente de desenvolvimento de aplicações em arquitecturas compartimentadas funcional. Este ambiente de desenvolvimento deve permitir realizar a construção das aplicações de cada partição de forma independente e a sua posterior integração, tendo em conta que num cenário típico estas são fornecidas por diferentes equipas de desenvolvimento ou fornecedores de software.

A arquitectura AIR (ARINC 653 In Space Real-Time Operating System), desenvolvida para responder aos interesses da indústria aeroespacial, particularmente da Agência Espacial Europeia (ESA), emerge como uma arquitectura compartimentada para o desenvolvimento e execução de aplicações aeroespaciais aplicando os conceitos de compartimentação temporal e espacial, contemplando os requisitos de tempo-real e segurança na operação. A arquitectura AIR permite a execução de sistemas operativos, tanto de tempo-real como genéricos, em partições independentes respeitando a norma ARINC 653, assegura independência da infraestrutura de processamento, e possibilita a verificação e validação independente dos componentes de software. O processo de verificação e validação assegura que o software cumpre com as suas especificações e responde aos requisitos definidos para a missão.

Durante o decorrer de uma missão espacial, podem surgir situações nas quais pode ser necessário, útil ou mesmo primordial modificar funções já existentes ou introduzir novas funcionalidades no computador de bordo de uma nave espacial para lidar com eventos inesperados. Para além disso, por exemplo na presença de falhas de alguns componentes, pode revelar-se útil proceder a alterações no plano de missão, tais como reconfigurar o escalonamento das aplicações em execução ou efectuar modificações nas aplicações existentes. Exemplos reais evidenciam a utilidade da realização de actualizações de software sem implicar a ocorrência de quebras de serviço ou paragens do sistema. Neste sentido, é preciso explorar abordagens que consigam concretizar as actualizações de componentes de software das partições em arquitecturas compartimentadas e ao mesmo tempo garantir que todos os requisitos dos sistemas críticos são cumpridos. É por isso muito importante que, para além da existência de uma plataforma adequada para o desenvolvimento e integração de aplicações, seja possível concretizar a transferência de actualizações das aplicações durante o normal funcionamento do sistema, sem que isso implique colocar em causa a segurança na operação e a pontualidade das aplicações em execução. Assim, o desenvolvimento de uma metodologia para actualização de componentes de software deve permitir que as alterações ao software possam ser aplicadas continuando a cumprir os requisitos temporais definidos para as restantes aplicações e a segurança no funcionamento.

Para efectuar actualizações de software assegurando os requisitos descritos, é necessário verificar quais as dependências que podem existir entre os diversos componentes do sistema, determinar quais os instantes em que as actualizações podem ser efectuadas, e concretizar, efectivamente, a substituição dos componentes. Isto requer um ambiente e ferramentas de desenvolvimento adequados, incluindo o suporte para a verificação e validação dos componentes modificados, durante o processo de construção e integração de aplicações.

Esta tese aborda soluções que, suportadas por métodos e ferramentas de desenvolvimento apropriadas, permitem proceder à actualização de aplicações em partições específicas e de tabelas de escalonamento de partições, tendo em conta os conceitos de separação espacial impostos pela norma ARINC 653, no que diz respeito à existência de mecanismos de protecção do acesso ao espaço de endereçamento das partições, e os requisitos temporais das mesmas.

Durante o decurso do trabalho desta dissertação foi desenvolvida uma metodologia para realizar actualizações para arquitecturas compartimentadas aplicadas aos sistemas aeroespaciais. Esta metodologia serviu de base para a concretização de duas funcionalidades essenciais. A primeira diz respeito à actualização de tabelas de escalonamento de partições, de forma a contribuir para a reconfigurabilidade do sistema, tornando disponível novas tabelas de escalonamento de partições construídas a partir de novos planos de missão. A segunda funcionalidade desenvolvida permite realizar a actualização de componentes de software das partições, permitindo assim adaptar as funções do veículo espacial durante a missão. Foi concretizado um protótipo para a demonstração das funcionalidades de actualização descritas no contexto da arquitectura AIR, acrescentando assim um novo mecanismo para atingir adaptabilidade à arquitectura. O melhoramento da arquitectura AIR com a possibilidade de realizar actualizações introduziu alterações no ambiente e ferramentas de desenvolvimento. Neste sentido, o processo de desenvolvimento de software em sistemas compartimentados foi revisitado, com vista à definição de um processo de construção e integração de aplicações que possibilite a concretização flexível de actualizações de partições e respectivos requisitos temporais.

O trabalho descrito nesta tese consistiu na análise dos problemas em aberto relacionados com a actualização de componentes de software e tabelas de escalonamento de partições, e no desenho e construção de soluções que permitem realizar actualizações durante o funcionamento do sistema. As soluções abordadas obedecem aos conceitos da segregação temporal e espacial das arquitecturas compartimentadas presentes nos sistemas aeroespaciais.

Futuros desenvolvimentos dizem respeito à actualização de partições dinamicamente, i. e., modificar os componentes de software sem que a partição correspondente tenha de ser parada. Isto pressupõe o estudo e desenvolvimento de serviços de gestão dinâmica de memória, e a sua adaptação às arquitecturas compartimentadas. A actualização de parâmetros de configuação e controlo do sistema, o estudo e aplicação de metodologias de actualização dinâmica de sistemas operativos em arquitecturas compartimentadas estão também previstos.

[1]Em cumprimentos do disposto no Artigo 27.º, n.º3, da Deliberação n.º1506/2006 Regulamento de Estudos Pós-Graduados da Universidade de Lisboa, de 30 de Outubro

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

AIR	ARINC 653 in Space RTOS
AOCS	Attitude and Orbit Control Subsystem
APEX	In ARINC 653 and AIR technology: Application Executive
ARINC	Aeronautical Radio, INCorporated
BOL	Beginning-of-life
CAN	Controller Area Network
CC	Cyclomatic Complexity
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DRAM	Dynamic Random-Access Memory
E^{2}PROM	Electrically Erasable Programmable Read-Only Memory
EOL	End-of-life
EPROM	Erasable Programmable Read-Only Memory
EPS	Electrical Power Subsystem
ESA	European Space Agency
FDIR	Failure Detection, Isolation and Recovery
GNU	GNU's Not UNIX
HM	In AIR technology: Health Monitor
I/O	Input/Output
IEC	International Electrotechnical Commission
IMA	Integrated Modular Avionics
IMADE	Integrated Modular Avionics Development Environment
ISS	International Space Station
KiB	Kibibyte

LEO	Low Earth Orbit
MCC	Mission Control Center
MECH	Structure and Mechanism
MiB	Mebibyte
MIPS	Million Instructions Per Second
MISRA	Motor Industry Software Reliability Association
MTF	Major Time Frame
NASA	National Aeronautics and Space Administration
OBC	Onboard Computer
OBDH	Onboard Data Handling
OI	Orbit Insertion
OO	On-orbit
OOS	On-orbit Servicing
OS	Operating System
PAL	In AIR technology: POS Adaptation Layer
PC	Personal Computer
PMK	In AIR technology: Partition Management Kernel
POS	In AIR technology: Partition Operating System
POSIX	Portable Operating System Interface (for Unix)
PST	Partition Scheduling Table
RAM	Random-Access Memory
RF	Radio Frequency
RISC	Reduced Instruction Set Computing
ROM	Read-Only Memory
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	Real-Time Operating System
SDRAM	Synchronous Dynamic Random-Access Memory
SLOC	Source Lines of Source Code
SPARC	Scalable Processor Architecture
SRAM	Static Random-Access Memory
SWaP	Size, Weight and Power consumption

TCS	Thermal Control Subsystem
TSP	Time and Space Partitioning
TTC	Telemetry, Tracking and Command
V & V	Verification and Validation
WCET	Worst-Case Execution Time
XAPEX	In AIR technology: Extended APEX
XML	Extensible Markup Language

Chapter 1

Introduction

Future space missions call for a new generation of space vehicles. This need has led to the interest of space agencies and industrial partners in defining the building blocks for new onboard computational platforms, where strict requirements of reliability, timeliness, safety and security are combined with a global requisite to decrease the size, weight and power consumption (SWaP) of the computational infrastructure.

The definition of partitioned architectures implementing the logical contention of applications in criticality domains, named partitions, allows to host different partitions in the same computational infrastructure and meet the enumerated requirements [1]. The notion of time and space partitioning (TSP) ensures that the execution of applications in one partition does not affect other partitions' timing requisites and that separated addressing spaces are assigned to different partitions [2].

The design of the AIR (ARINC 653 in Space Real-Time Operating System) Technology has been prompted by the interest of the space industry partners, especially the European Space Agency (ESA), in applying the TSP concepts to the aerospace domain [3, 4]. The AIR architecture allows the execution of both real-time and generic operating systems in distinct partitions, ensures independence from the processing infrastructure, and enables independent development, validation and verification (V & V) of software components.

The computational infrastructure of a typical spacecraft hosts several subsystems, consisting of avionics functions, such as Attitude and Orbit Control Subsystem (AOCS), Onboard Data Handling (OBDH) and Telemetry, Tracking and Command (TTC), and payload, which closely interact with each other. In TSP systems, the several functions share the same computational resources being hosted in different partitions, and supported by embedded systems.

The writing, build, debug and maintenance of applications in partitioned architectures requires overcoming some limitations imposed by the standard development tools, that were not designed to the development of such applications. It is necessary to adapt the existing tools in order to have an environment suitable to develop applications for partitioned architectures.

Furthermore, due to unexpected events or the change of environmental conditions, it may be necessary to improve the functions hosted in a spacecraft or even to add new ones, during a mission. This way, the onboard computing systems should be able to handle the reconfiguration and update of spacecraft functions without compromising the timeliness and the safety of the system.

This thesis describes a methodology for reconfigure and update system components and applications for time- and space-partitioned architectures, applied to aerospace systems, motivated by the need to adapt system behaviour to changing conditions or unexpected events during the system operation. The proof-of-concept prototype will be implemented within the scope of the AIR architecture. This involves improving the AIR architecture with the support for the remote modification of system's parameters and the update of software onboard components.

1.1 Motivation

Throughout time, space exploration experience has demonstrated that recovery from many severe failures can be achieved through the update of software components [5]. The need for in-flight programmability has been identified by space agencies [6]. The incentive to build a system that supports the system reconfiguration, or even the modification of software components, comes from several incidents in the scope of aerospace missions, almost always related to failure events. The software upgradeability is a crucial property to spacecraft survival in long-term missions [7].

Mars Pathfinder success history

During the course of a mission, there may appear situations on which it could be necessary, useful and even primordial to change the actual system configuration. An anthology example is the case of the NASA's Mars Pathfinder mission [8]. The rover Pathfinder, after landing on Mars surface and start collecting meteorological data, began suffering of system restarts constantly. The failure in the system was identified has a classic priority inversion problem. The team working in Mars Pathfinder mission identified the reason of the system restarts through event log analysis and debug capabilities that fortunately were enabled. Then, to solve the problem, it was built a program to change the value of the global variables that were causing the malfunction. The program was uploaded to the rover and executed, turning the system back to its normal execution state.

Without the ability to debug and modify system parameters during its execution, Mars Pathfinder mission could have failed because the problems would have not been resolved. Instead of that, this mission and the onboard software system form a success story due to the way that the problem was identified and solved.

NASA's Mars rover Spirit now working as a stationary research platform

On a similar way, in the presence of events, such as component failures, it may be useful to change the mission plan, which may involve, for instance, the redefinition of the scheduling of applications or the modification and inclusion of new features.

NASA's rover Spirit was sent to Mars in 2003 to, with its twin rover Opportunity, continue the work started with the Mars Pathfinder mission, having as the major purpose the exploration of Mars surface and geology. At May 2009, Spirit become stuck on a soft sand terrain and after several months of trying without success to release the rover and put it back to its original path, the mission plans were changed [9]. NASA's team on Earth decided that Spirit would work as a stationary research platform and thus contributing on such a way that would been impossible to a mobile platform, such as detecting small oscillations on the rotation of the planet that could foresee the presence of a liquid core on it. The system capability to change the mission's payload was essential to continue to use the rover, albeit in a different manner.

More recently, the efforts in assigning a new mission plan to the rover revealed to be fruitful, since the analysis of the results over the research performed by the Spirit demonstrated the presence of water in Mars [10].

1.2 Goals and contributions

The work presented in this thesis concerns improving a platform for the development of applications for partitioned architectures; the analysis of the open research questions related to the update of software components and system configurations, and; the design and development of solutions to allow performing updates during the system execution, obeying to the segregation concepts of time- and space-partitioned architectures aiming aerospace systems.

The development of this thesis involves:

- Understanding the importance of performing software updates in spacecraft and modifying system control parameters, during the course of a mission.

- The definition of an update methodology for time- and space-partitioned architectures, concerning the reconfiguration of the scheduling of the system applications at execution time, and the update of application software components hosted in partitions.
- The implementation and integration of these features in the AIR architecture, the evaluation, and the discussion of the obtained results.

1.3 Institutional context

The development of this thesis took place at the Large-Scale Informatics Systems Laboratory (LaSIGE-FCUL), a research unit of the Informatics Department (DI) of the University of Lisbon, Faculty of Sciences. This work was developed within the scope of the AIR-II (ARINC 653 in Space RTOS – Industrial Initiative) project[1], which fits in the Timeliness and Adaptation in Dependable Systems research line of the Navigators group. The author of this thesis integrated the LaSIGE-FCUL AIR-II team as a junior researcher.

1.4 Publications

There were produced several articles in the scope of the AIR-II project, some of them presenting preliminary work on the subject approached in this thesis, and the remaining resulting of the work herein described.

The following papers were published in international and national conferences:

1. **J. Rosa**, J. Craveiro, and J. Rufino, "Safe Online Reconfiguration of Time- and Space-Partitioned Systems", in *9th IEEE International Conference on Industrial Informatics (INDIN'2011)*, Caparica, Lisboa, Portugal, Jul. 2011, accepted for publication. [11]
2. **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. [12]
3. **J. Rosa**, J. Craveiro and J. Rufino, "Exploiting AIR Composability towards Spacecraft Onboard Software Update", in *Actas do INForum 2010, Simpósio de Informática*, Braga, Portugal, Sep. 2010. [13]

[1] http://air.di.fc.ul.pt/air-ii

The following AIR-II technical reports, relevant to this thesis, were produced:

1. J. Rufino, **J. Rosa** and J. Craveiro, "Desenvolvimento e actualização de software para sistemas aeroespaciais em arquitecturas compartimentadas", AIR-II Technical Report RT-10-10, Oct. 2010. [14]

2. **J. Rosa**, J. Craveiro, and J. Rufino, "Challenges in the Design and Development of Spacecraft Onboard Software Update", AIR-II Technical Report RT-10-12, Nov. 2010. [15]

1.5 Document outline

The remainder of this document is organized as follows:

Chapter 2 Concerns the important concepts to the unroll of this thesis. This includes a brief description of spacecraft systems and components as well as onboard computing platforms and software; a summary of the common faults in spacecraft during the past thirty years along with some statistical results, and examples of missions that involved spacecraft failures; issues concerning the verification and validation of critical applications, and; analysis of software update methodologies.

Chapter 3 Presents the AIR architecture. It describes the properties inherent to the AIR architecture, including schedulability and composability, and explains the application development process.

Chapter 4 Defines the challenges approached in this thesis. This involves the description of the computational model used; the definition of open problems and issues in onboard software update, and; a brief discussion featuring possible solutions to onboard software update.

Chapter 5 Addresses the reconfiguration of the scheduling of system applications at execution time, presenting an algorithm for the safe update of application schedules and its complexity analysis; a proof-of-concept prototype, and; relevant results.

Chapter 6 Addresses the update of application software components hosted in partitions that can be shut down, without significant functional impact. This includes the presentation of an algorithm for update application software components; the proof-of-concept prototype, and; relevant results.

Chapter 7 Presents some concluding remarks of the work approached in this thesis and highlights future work developments.

Chapter 2

Background

This chapter starts with a description of spacecraft subsystems, probing the on-board computational infrastructure, thus giving an information background required to understand the issues addressed in this thesis. Then, common failures in spacecraft are presented along with statistical data, motivating the need for online[1] reconfiguration and software update. This leads to the necessity to understand the space mission requirements and adopt strict verification and validation (V & V) measures, which will also be addressed. Finally, this chapter exposes methodologies for software update described in the literature.

2.1 Spacecraft systems overview

A spacecraft consists of several individual subsystems, which closely interact with each other[2]. Spacecraft can be divided in two principal elements, the payload and the avionics functions. The payload consists of scientific instruments and experiments, being the motivation for the mission. Avionics may consist of the following spacecraft subsystems, here described briefly:

Communications Subsystem Provides an interface with radio frequency (RF) systems and antennas, allowing the data transfer between the space vehicle and the ground control. The data transfer is supported by (secure) communication protocols. The communications subsystem includes a non-executive command detection component responsible for receiving and passing all the commands originating from the ground control to the Telemetry, Tracking and Command (TTC) subsystem, which interprets and executes them [16].

[1]In the scope of this thesis, "online" or "in-field" means performing certain task during the operational phase of a spacecraft, i. e., during a mission.

[2]The subsystems physical distribution and implementation may vary considerably between spacecraft, as well as the designations assigned to each subsystem described in this work.

Telemetry, Tracking and Command Subsystem (TTC) Executes the spacecraft telemetry, telecommand and control functions. It receives and processes commands to control the spacecraft and to operate the payload, as well as housekeeping and science data originating from the ground control (passed through the communications subsystem) or other spacecraft subsystems [17, 18].

Figure 2.1 presents the common spacecraft subsystems, highlighting the subsystems relevant to follow the development of this thesis. These, represented in the top of the figure, correspond to the subsystems responsible for the space vehicle control and can be somehow managed by software. At the bottom are represented the subsystems associated in the first instance to the mechanical part of the spacecraft. This representation of spacecraft organization was adapted from Magellan space flight system functional block diagram that can be found in [17].

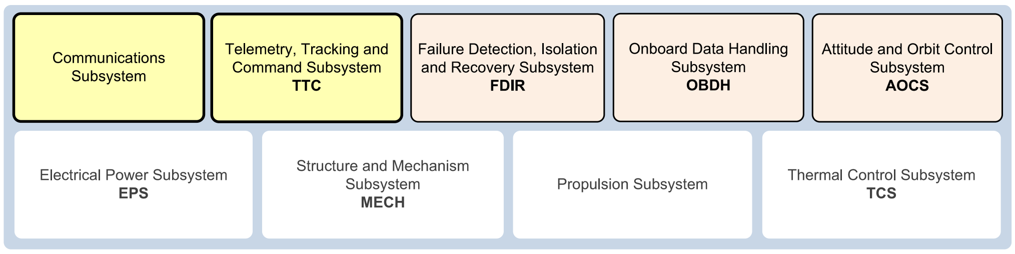

Communications Subsystem
Telemetry, Tracking and Command Subsystem TTC
Failure Detection, Isolation and Recovery Subsystem FDIR
Onboard Data Handling Subsystem OBDH
Attitude and Orbit Control Subsystem AOCS
Electrical Power Subsystem EPS
Structure and Mechanism Subsystem MECH
Propulsion Subsystem
Thermal Control Subsystem TCS

Figure 2.1: Structural organization of spacecraft subsystems

The remaining subsystems included in spacecraft, although with no direct relevance to this work, are the following:

Failure Detection, Isolation and Recovery Subsystem (FDIR) Is responsible for switch between onboard components in the presence of failures and ensure that a failure with origin in a certain component, such as the payload, does not propagate to other spacecraft subsystems. The main goal of FDIR is to effectively detect faults and accurately isolate them to a failed component in the shortest time possible [19]. If a single failure is detected and can be isolated, FDIR switches to a redundant component, marking the failed one as unhealthy [20].

Onboard Data Handling Subsystem (OBDH) Is responsible for collect, store, process and format spacecraft housekeeping and mission data for downlink or to be used by onboard computers and other spacecraft subsystems [16, 5].

Attitude and Orbit Control Subsystem (AOCS) Has as its prime purpose stabilize the main structure of the spacecraft correctly and orient it in desired directions during the mission [18]. This requires the vehicle to determine its attitude, using sensors, and control it, using actuators (e. g., reaction wheels; thrusters) [16]. It monitors and modifies the spacecraft attitude and trajectory to meet mission objectives despite disturbances during on-orbit operations.

Electrical Power Subsystem (EPS) Generates and stores all of the spacecraft's power, feeding the other spacecraft subsystems. Solar panels are used to derive electricity from the sunlight [18, 21].

Structure and Mechanism Subsystem (MECH) Supports the mechanics of the other spacecraft subsystems [5].

Propulsion Subsystem Is composed by rocket engines and thrusters that are used to accelerate spacecraft, make trajectory correction maneuvers, and maintain the spacecraft in its orbit [22].

Thermal Control Subsystem (TCS) Provides reliable temperature control to accommodate variations in the spacecraft heat load thus maintaining the temperatures of each component on the spacecraft within their allowable limits [21].

The launch vehicle plays an important role in a mission as it carries the spacecraft through the Earth's atmosphere and places it in orbit. Manned spacecraft must also include a life support subsystem, which consists of a group of devices that allow a human being to survive in space. The mission management and remote support to space vehicles is done on Earth by the Mission Control Center (MCC), commonly referred to as ground segment. The ground segment is sometimes seen as a spacecraft component as it is vital to its operation [18].

A complete description of all spacecraft subsystems including their components and operations can be found in [18]. Design of spacecraft and mission analysis aspects are addressed in [16].

2.2 Onboard computers

The space technology has evolved greatly over the past decades. Besides the notable advances in energy-conversion technologies as well as in many other areas, the electronic computers and software have occupied the first places during this period [18]. The space industry has quickly assimilated the emerging technology, which revolutionized the autonomy and flexibility of spacecraft, and allowed to turn a potential mission failure into a grand success [8, 22, 23].

The production of the onboard software follows formal documentation such as national and international standards, hardware/software control documents, specifications and user manuals. Code reuse in different missions and spacecraft has turned to be a common practice. The missions are defined by advanced software due to its complexity, and the flight software is developed following approaches identical to those used to develop terrestrial control applications [18].

Onboard computers (OBCs) are designed to accomplish requirements for reliability, complex data organization, autonomous decision-making, intensive signal processing and multitasking [24].

Spacecraft should have a high level of autonomy since they may spend some time out of range of any ground station[3]. For example, Low Earth Orbit (LEO) polar orbiter satellites spend only one hour per day over a mid-latitude ground station [18]. This way, the investment on onboard software should not be underestimated.

In the past, spacecraft were designed to have OBCs, i. e., CPUs and microcontrollers, in all major subsystems and payloads, being each OBC responsible for a spacecraft function [25]. Although, a new generation of space vehicles, enabling reduced size, weight and power consumption (SWaP), has been adopted, and nowadays typical spacecraft have a single OBC to handle with all computational functions. The transition from federated architectures to integrated modular avionics (IMA) motivated the use of time- and space-partitioned (TSP) architectures in spacecraft computational systems, to implement the logical separation of each spacecraft function [26, 2].

The CPUs used in spacecraft must be highly reliable and very durable. Actually, spacecraft designers opt to use microprocessors that have been largely tried and tested (according to the MIL-STD-883 standard, for instance, to insure reliable operation [27]), instead of using the latest and greatest chips. Spacecraft OBCs use 32-bit or 64-bit microprocessors programmed using secure and certified methods and tools. Microprocessor programs are written in languages such as C, C++ or ADA.

Table 2.1 presents some examples of CPUs used in different spacecraft [25, 28]. Some spacecraft use radiation-tolerant versions of processors found in common computers whereas others employ processors especially developed for space use such as those from the SPARC LEON family [29]. Figure 2.2, adapted from [30], represents the evolution of onboard microprocessors used in (European and North American) space vehicles over the years, presenting the values for million instructions per second (MIPS) and typical Random-Access Memory (RAM) chip size for a type of processor. This graphic is complemented with the program-

[3]A ground station is a terrestrial terminal handling telecommunications with spacecraft.

ming languages of the flight software and the type of real-time operating systems (RTOS) used.

The interaction between the several spacecraft subsystems may be done using controlled area network (CAN) buses. The processing power and storage available onboard follows those found on Earth, however, techniques of data compression are used in order to spare as much storage space as possible [18]. Typically, data is stored in RAM or Flash solid-state memories, operating on a block or file basis. Read-Only Memory (ROM) is built into the OBC before launch and contains the basic instructions and safe guard modes during operations [17].

Table 2.1: CPUs used in spacecraft

Spacecraft and launch year	**CPU description**
Viking (1976)	RCA 1802 (built with Silicon-on-Sapphire which is much more stable in a radiation environment)
Voyager 1 and 2 (1977)	RCA 1802
Space Shuttle (1981)	Intel 8086 and RCA 1802, later Intel 80386 (uses the APA-101S computer)
Galileo (1989)	RCA 1802
Hubble Space Telescope (1990)	Originally a DF-224 (8-bit) and now a 80486
Pathfinder (1996)	BAE RAD6000
International Space Station (ISS) (1998)	Intel 80386SX-20 w/ Intel 80387 (there are several computers on the ISS. The most important are the common computers which use the i386)
Cluster (2000)	1750A (MIL-STD 16-bit non-RISC CPU)
Spirit and Opportunity Rovers (2003)	BAE RAD6000
SMART-1 (2003)	ERC32 (version of SPARC V7 enabling the use of commercial software)
Rosetta (2004)	1750A

Onboard memory types and typical capacity can be classified as follows:

EPROM (Erasable Programmable Read-Only Memory) Hosts the board boot and initialization software (typically 16 – 256 KiB[4]).

E^2PROM (Electrically Erasable Programmable Read-Only Memory) / Flash Hosts the mission software and boot container (typically 2 – 8 MiB[4]).

SRAM (Static Random-Access Memory) Contains the executing software and variables (typically 4 – 16 MiB).

DRAM (Dynamic Random-Access Memory) Is typically used for store large amounts of data (typically 16 – 512 MiB).

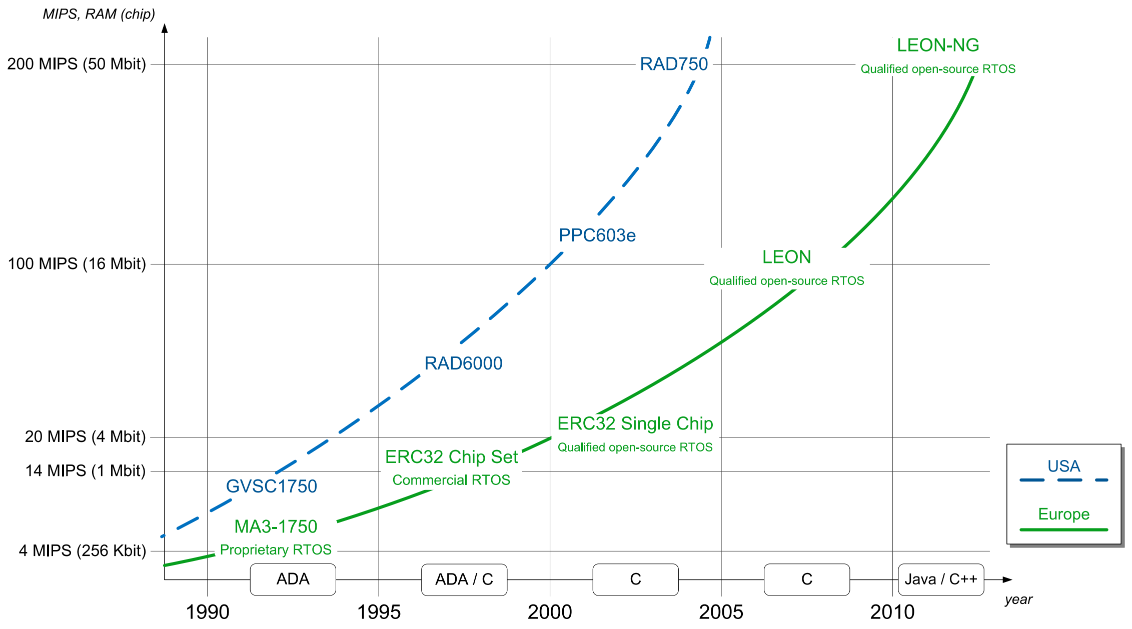


Figure 2.2: Onboard processor evolution

Onboard data storage, performed by the OBDH subsystem, is achieved using solid-state memories, namely Flash memories [30]. Data storage is usually a required feature in spacecraft computational infrastructure since the data collecting rate and volume could exceed the capacity of the link with the ground segment, as well as its availability [5, 18]. Data coding and error correction techniques are used to maintain the integrity of the data. The OBC memory is organized in the same way that is done in storage disks found in terrestrial computers, with data arranged hierarchically and files identified by name.

[4] 1 kibibyte (KiB) = 2^{10} bytes = 1024 bytes; 1 mebibyte (MiB) = 2^{20} bytes = 1024 kibibytes. This corresponds to the prefixes for binary multiples defined in the IEC 60027-2 standard specification [31]

Spacecraft flight software is integrated in high reliable and robust real-time operating systems, such as Wind River VxWorks[5] or Real-Time Executive for Multiprocessor Systems[6] (RTEMS). RTEMS is particularly interesting for use in space onboard software systems given its qualification for spaceborne applications [32]. Flight software associated to a certain spacecraft function, such as AOCS or TTC, usually does not exceed few hundreds of KiB [33].

The components of a space vehicle computational infrastructure described throughout this section are illustrated in Fig. 2.3. This scheme embraces a spacecraft structural organization where a single CPU handles the execution of several spacecraft functions.

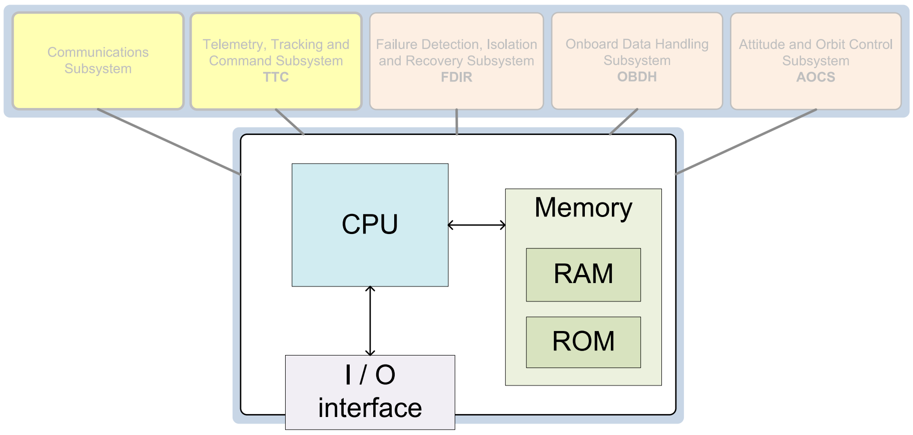

Communications Subsystem
Telemetry, Tracking and Command Subsystem TTC
Failure Detection, Isolation and Recovery Subsystem FDIR
Onboard Data Handling Subsystem OBDH
Attitude and Orbit Control Subsystem AOCS
CPU
Memory
RAM
ROM
I / O interface

Figure 2.3: Simplified onboard computational infrastructure based on a unique CPU

As long as the spacecraft is visible and available from Earth it is tracked by real-time software operating in the associated ground station. With the ability to access the several spacecraft OBCs from ground segment, one may calibrate the spacecraft parameters, adjust the control algorithms and upload new functions or control software to enhance performance or adapt to new requirements. Nowadays, there is a trend in space industry to adopt commercial off-the-shelf (COTS) products in ground stations in order to reduce costs. Concerning the spacecraft equipment, the adoption of COTS components is also related to achieving better performance results. Although, the interaction with standard components requires additional interface mechanisms [34].

Unlike past practice, designing spacecraft computational systems supporting advanced interactions with the ground segment, which may include operations such as the modification and debug of the flight software, may prove to be a measure of great importance particularly with regard to overcome the occurrence of failures [18, 5].

[5]http://www.windriver.com/products/vxworks/
[6]http://www.rtems.com/

2.3 Common faults in spacecraft

During the course of a mission, it may be useful or even necessary to introduce new functions or modify existing ones in order to deal with unexpected events. An early example where such features had an essential role was the NASA's Apollo 14 mission [35]. Prior to the descent to Moon's surface, the Antares lunar excursion module had a serious problem related to a faulty abort switch. The erratic behaviour of the switch could cause the onboard computer to order the spacecraft to climb back into orbit. The solution approached to overcome this issue involved reprogramming the flight software to ignore the false abort command. The software modifications were transmitted via voice communication and the changes were manually entered by the spacecraft crew. These kind of episodes in the history of space missions highlighted the importance of reprogrammability as a required property in space systems [33].

Along with the rising processing power and complexity of OBCs, the potential for software errors leading to spacecraft failures increased. The causes of software failures may be systematic, however the faults occur randomly in time. Typically, the software errors are originated by (i) inadequate system and software engineering, (ii) inadequate review activities, (iii) ineffective system safety engineering, (iv) inadequate human factors engineering, and (v) flaws in the test and simulation environments [36, 18].

Failures in space missions

The study and analysis on spacecraft failures done in [5], based on information retrieved from various sources, over a set of on-orbit spacecraft failures occurred in 129 spacecraft between 1980 and 2005, revealed that many spacecraft suffer unrecoverable failures in early times after their launch and others, despite the occurrence of several failures, exceed their expected lifetime after applying failure recovery procedures. The purpose of the work described in [5] was to estimate the impact of failures on the mission and identify the critical subsystems and recurrent failure modes. The failure analysis fell over the following subsystems groups: AOCS; EPS; OBDH and TTC; MECH, payload and some other miscellaneous subsystems. The results revealed that the software failures represented a small percentage of the overall spacecraft failures when comparing to the failures caused by electrical and mechanical spacecraft components.

The spacecraft failures caused by software errors, although, can be more easily fixed if the system supports software patching and modification. For example, during the first European lunar mission, ESA SMART-1, software modules controlling the electric propulsion which have suffered anomalies, for instance in the error detection and correction algorithm, were subjected to adjustments and onboard software patching in order to overcome the adverse situation [22].

With respect to the impact of spacecraft failures in the mission itself, nearly 40% of the failures analysed resulted in catastrophic events, such as the loss of the mission, and the percentage of mission degradation rises to about 65% [5]. This supports the recommendations for the development of more flexible aerospace systems and the adoption of trusted software verification and validation (V & V) techniques [5, 16]. The SMART-1 mission benefited greatly from the spacecraft capabilities to handle software modifications and parameter tunings. Nonetheless, not all the parameters offered flexibility, which caused a negative impact in the mission in specific situations. For example, a malfunction of sensitivity to radiation in EPS inducing shutdowns could have been solved modifying the software not to trigger the alarm, thus avoiding the EPS shutdown and the waiting time to restart [22].

In the graphic of Fig. 2.4 are represented the percentages of the most significant spacecraft subsystems affected, as they appear in [5]. AOCS is the subsystem most affected (32%). Most of the AOCS anomalies are caused by environmental conditions and can be overcome via software patches [22]. AOCS is followed by the EPS (27%), OBDH (15%), TTC (12%) and other spacecraft subsystems (the remaining 14%).

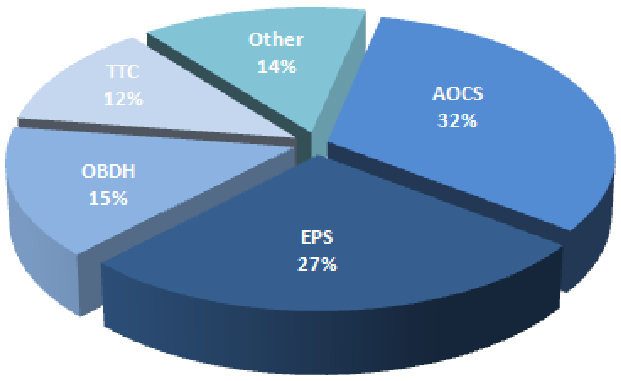


Figure 2.4: Percentage of subsystems affected during a mission

Temporal considerations

A space mission is composed by several phases. The launch phase concerns all the preparations for spacecraft launch and the launch itself. The failures occurred during this phase are the launch failures. After the launch, the spacecraft enters the cruise phase where are performed diverse operations such as real-time commanding, spacecraft tracking and monitoring, or preparation for encounter.

The encounter phase includes all the operations related to planetary orbit insertion, descent and landing, and sampling, for example. Depending on the state of spacecraft health and mission funding, the mission cruise and encounter phases may be extended [17]. Failures occurred during the operational phases can be grouped in beginning-of-life (BOL) failures; orbit insertion (OI), either Earth or distant planets and on-orbit (OO) failures, and; end-of-life (EOL) failures [5, 37].

BOL failures include the in-orbit checkout. OI failures happens during the spacecraft orbit insertion maneuver, i. e., deceleration or acceleration maneuvers to allow the spacecraft to be captured into orbit. OO failures occur during spacecraft operational events while on orbit. EOL failures correspond to the spacecraft consumable depletion or component obsolescence.

The time to failure is here classified as follows. BOL failures concern all failures occurred in the first year of the mission, that correspond to 41% of all failures, suggesting insufficient testing or inadequate modeling. OI and OO failures are those occurred between the second and the eighth year, totaling 53%. EOL failures happen from the eighth year forward, corresponding to only 6% of all failures [5]. The graphic of Fig. 2.5 shows the relation between the percentage of failures and the time to failure after the spacecraft launch.

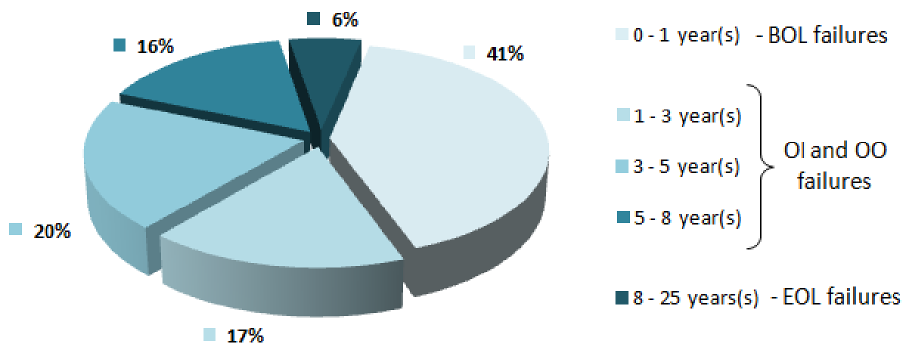


Figure 2.5: Time to failure after launch

Spacecraft failures analysis throughout the years

Spacecraft failures analysis throughout the years has demonstrated that the failures rate can be mitigated either by the use of software patches or through on-orbit servicing (OOS), i. e., maintenance, upgrade and repair of equipment. The graphic illustrated in Fig. 2.6 compares the average per year in the overall study time (1981–2000) and the period during 1996–2000, according to [37]. It can be observed an increase of the number of failures in the period 1996–2000, with the exception of the OI failures, were this number remains approximately constant. EOL failures were not documented. The high failure rates during the period of

1996–2000 may be justified with the existence of more extensive and accurate failure reporting methods and tools, comparing to earlier dates, as suggested in [37].

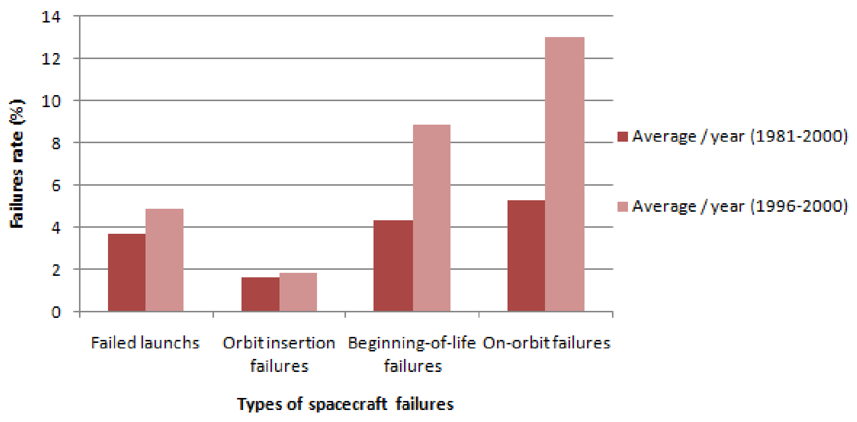


Figure 2.6: Spacecraft failures during 1981–2000

2.4 Verification and validation process

In the scope of spaceborne systems and applications, the verification and validation (V & V) process ensures that the developed software meets its specifications and satisfies the requirements defined for the mission. The in-field V & V of system configurations and software components have a remarkable importance to determine whether is safe to proceed with update operations and reconfigurations. There are different approaches with regard to the V & V process [38, 39], although the majority of them agrees that the verification and validation should be performed during all stages of the development process. Furthermore, it may be required to extend the V & V process also to the product's operational lifetime featuring, for instance, the possibility to perform additional tests or to verify the correctness of in-flight software updates. This implies that the spacecraft's computational system should somehow support the online verification of software components, to ensure the safety and robustness of the system [40, 41].

Shortcomings in requirements development and the lack of a strong V & V process can cause failures either individually, collectively, or in conjunction with other faults. For example, in Apollo 13 mission, the operating voltage of the thermostatic switches for the heaters of the oxygen tanks were modified without changing the voltage specification and testing the new value on the switches. This was a configuration management failure that should be detected by verification. However, the mission was a success because the lunar module, the crew and the backup systems were robust [38]. Another famous example is Space Shuttle Challenger. The requirements for the mission seemed to be correct, however the design, manufacturing, testing, and operation were faulty. The low air temperature, which led to the explosion of Challenger, was below design expectations. The decision to launch in an environment for which the system was not designed was a validation mistake [38].

An adequate certification of software and qualification of tools is of extreme importance to meet the development requirements. Current practices of civil aviation, military aviation and aerospace industry, along with airborne systems and equipment guidelines remain vague, which can lead to different interpretations by the industry. The software developed for safety-critical systems must be supported by qualified tools and implemented using appropriate mechanisms provided by programming languages. Despite the popularity of C and C++ programming languages in the development of safety-critical systems, these languages include many features that are not suitable for this purpose, whereas Ada is well suited and meets the safety requirements [39]. Furthermore, some standards are not fully adapted for building safety-critical systems, such as the MISRA C standard, which aims to contribute to make C a safer language, although does not address all known fault models [42].

Besides the strict verification and validation applied to the spacecraft flight software on the ground, it is important to have safeguard mechanisms to verify the system parameters during the flight and prevent faults in unexpected situations, namely due to the difficulty of simulate the space environmental conditions on the ground. The onboard validation of software is crucial to ensure whether is safe to proceed with the system execution. This is specifically important after the modification or update of software components [40, 41].

The in-field safety validation may benefit from the use of contracting mechanisms based on formal methods [41]. This technique supports the verification of system reconfigurations and upgrades based on the analysis of the current system configuration. The methodology suggested in [41] consists on the requirements analysis of the application to be updated and on the validation of contracts during system execution, before admitting corresponding configurations to take effect in the system. This allows the modification of system parameters and the integration of software components to be verified autonomously by the system itself.

Since the inability to avoid several failures on spacecraft is related to the difficulty to simulate the space conditions on Earth, it is noteworthy the importance of having the possibility to perform tests under similar conditions. The work present in [43] focuses on test methodologies and simulation environments. The approach followed was designed to test software behaviour under disturbed conditions, simulating the space environmental conditions and attaining the greatest possible dependability of embedded software systems by reducing development errors and handling runtime anomalies. This work discusses the advantages of software-in-the-loop simulations over the hardware-in-the-loop simulation approach, this is, testing the real software carried out on a simulated environment versus the use of prototypes of the system under simulation.

2.5 Software update

This section addresses methodologies for system reconfiguration and software update. There are different approaches regarding the reconfiguration of aerospace applications. In airborne systems, those concern complex online or offline techniques to ensure the flight or mission's effectiveness. System reconfiguration can be achieved using methods such as multi-static reconfiguration, which consists of the activation of a predefined configuration selected autonomously according to the system health state [44]. The possibility to reconfigure autonomously a space vehicle in operation through adjustments of the system control parameters and algorithms, for example, is essential to its adaptation to different mission phases.

Moreover, the onboard software running on spacecraft OBCs should be upgradeable, i. e., the system should support the update or modification of software when required. There are diverse techniques and methodologies for software update. However, the requirements for software update methodologies in space vehicles are much more strict than those required for updating terrestrial computers. For instance, the updates should be executed in exact moments considered safe, their duration should be predictable and well specified, and they should not interfere with the system operation [45, 46]. The update methodology proposed in [45] preserves the original deadline guarantees, however it does not address what should be done when the timing requisites need to be modified. Software update methodologies may be built based on replication principles, such as the definition of two execution blocks to perform the updates [47].

There are several works in the literature concerning software update in the domain of real-time systems [46, 47, 48] including in the avionics industry [49], however with respect to performing updates in spaceborne systems, there is no research line established. With respect to the update of software in TSP-based systems, there were not found works covering this subject in the literature.

Techniques of dynamic software update, which refers to the modification of software components without the need to stop the current system execution, may be applied to update onboard software, although fulfilling the requirements for onboard update methodologies referred [50, 51]. An approach for dynamic update of applications in C-like languages is provided in [51] and focuses on the update of the code and data at predetermined times, allowing the online verification of the code's safety. The methodology suggested in [50] covers the timeliness requirements to achieve a safety environment for dynamic software update on real-time systems, running a COTS operating system. The methodology proposed requires the identification of specific points in time to perform the component's update while in [48] is proposed an approach to the update of real-time applications without any presumption about the application execution times.

2.6 Summary

This chapter grouped several aspects with relevance to the development of this thesis. It described spacecraft subsystems, focusing on the onboard computers; common spacecraft mission failures; verification and validation process, and; methodologies for software update.

The following chapter will address the AIR architecture, a time- and space-partitioned environment for the development and execution of aerospace applications, along with its inherent properties, and the applications development process.

Chapter 3

AIR Technology design

This chapter aims to present the AIR architecture, which will support the features proposed in this thesis, concerning the update of partition schedules and applications. Beyond a general description of the architecture, this chapter details the architecture components. It explains how is defined the AIR partition scheduling. Then, it presents the composability properties of the AIR architecture. Finally, this chapter exposes the AIR applications build and integration process.

3.1 System architecture

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), implying that the execution of applications in a partition does not affect other partitions' timeliness and that different partitions have independent addressing spaces. The AIR architecture allows applications to be executed in logical containers called partitions. 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 are guaranteed. The spatial partitioning relies on having dedicated addressing spaces for applications executing on different partitions [3, 4].

The AIR architecture, illustrated in Fig. 3.1, relies on the *AIR Partition Management Kernel* (PMK) to enforce robust TSP. An operating system, herein referred as *Partition Operating System* (POS), is provided per partition. It is foreseen the use of different operating systems among the partitions, either real-time operating systems (RTOS) or generic non-real-time ones. Each POS is wrapped by the *AIR POS Adaptation Layer* (PAL) hiding its particularities from other AIR components thus ensuring flexibility and independence in the integration of each POS kernel.

At the Application Software Layer (Fig. 3.1), applications consist of one or more processes, which make use of the services provided by an *Application Executive* (APEX) *interface*. In addition, a partition holding system functions, may invoke also specific functions provided by the POS, thus being allowed to bypass the standard APEX interface.

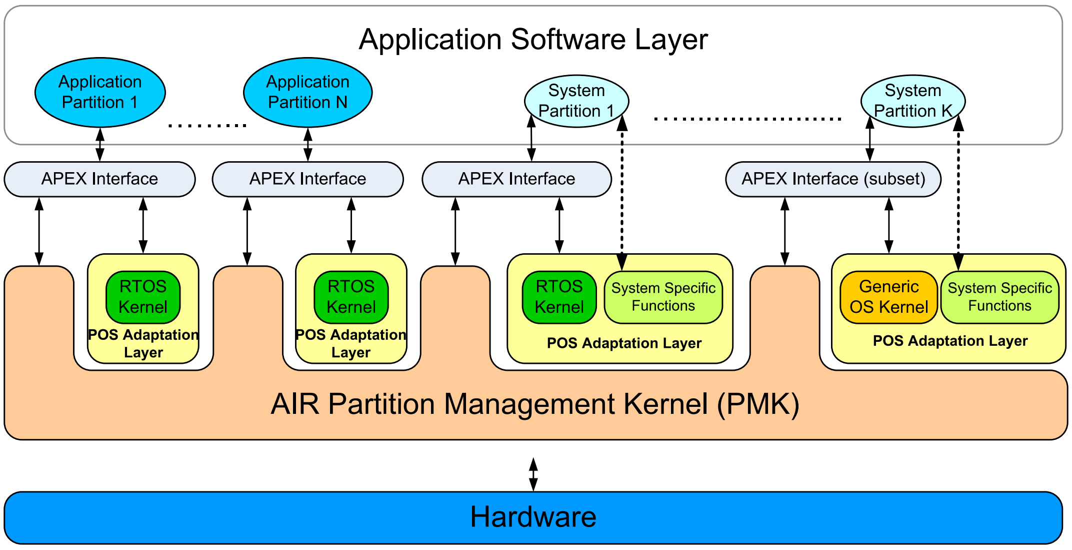


Figure 3.1: AIR system architecture and integration of partition operating systems

3.1.1 Portable Application Executive (APEX) interface

The *Application Executive* (APEX) *interface* component provides a standard programming interface with a service definition derived from the ARINC 653 specification [52]. 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 [53].

3.1.2 AIR Health Monitor

The AIR architecture incorporates a *Health Monitor* (HM) component which is responsible for handling and containing errors to their domains of occurrence. The action to be performed in the event of an error is defined by the application programmer through an appropriate error handler. This error handler is an application process that should include a systemwide reconfigurability logic, which comprises the redefinition of control parameters or the issue of a different schedule request, thus helping achieve system adaptability.

3.1.3 Interpartition communication

The organization of spacecraft software components in different partitions requires interpartition communication facilities, since a function hosted in a partition may need to exchange information with other partitions (e. g., the communications subsystem passes the commands issued by the ground mission control to the TTC subsystem, which in its turn proceeds with the suitable operations). Interpartition communication consists of the authorized transfer of information between partitions without violating spatial separation constraints [4].

3.2 Schedulability

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. 3.2).

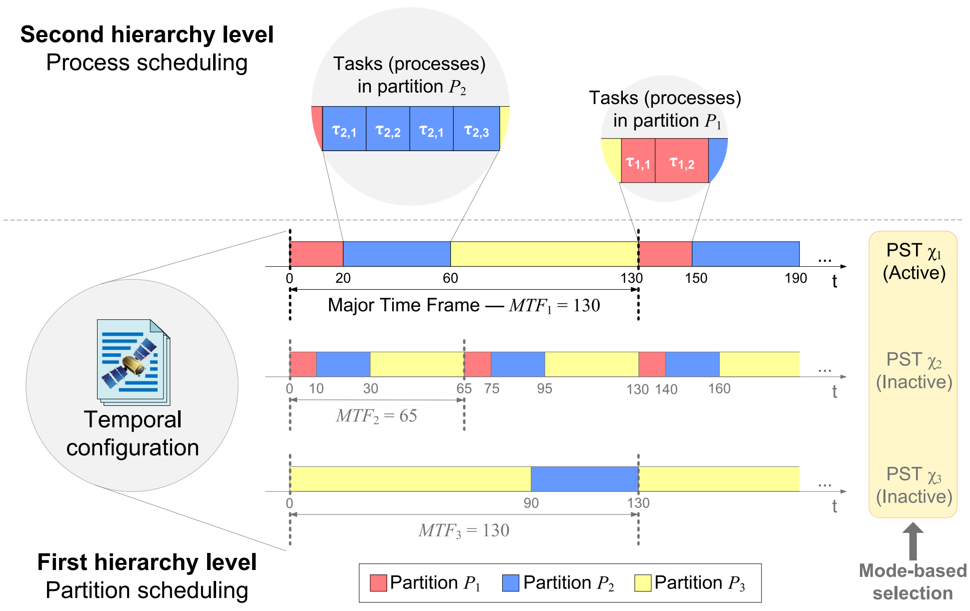


Figure 3.2: Two-level mode-based partition scheduling

The original ARINC 653 [52] notion of a single fixed partition scheduling table, defined offline, is limited in terms of timeliness control and fault tolerance. To address this limitation, the AIR Technology design incorporates the notion of *mode-based partition schedules* (Fig. 3.2), allowing the switch between different partition scheduling tables (PSTs) according to different mission phases or operating modes during the execution time, and regarding the accommodation of component failures [4, 54]. A schedule switch can be requested by a specifically authorized and certified partition through the invocation of an APEX primitive [39]. This can result from a command issued from ground mission control or from the reaction to environmental conditions. The *AIR Partition Scheduler* is responsible for guaranteeing to make a schedule switch effective at the end of the respective MTF.

3.3 Composability

The modularity of the AIR architecture design and of its build and integration process further enables the *composability* of AIR-based systems [55], in both time and space domains. Composability means properties established for individual components hold also after the components are assembled together into the system. With respect to the temporal domain, the use of a fixed cyclic partition scheduling scheme dictates that the timeliness guarantees of each partition are defined by the processing time assigned to each partition. In the spatial domain the composability properties ensure that the partition's memory and I/O resources are protected against unauthorized access from other partitions. The composability properties are thus inherent to the AIR modular architecture.

Composability allows the independent verification and validation of software components to different software developers during the build and integration process, facilitating the overall system certification. From the point of view of one partition's provider, this further signifies that the development and validation do not depend on knowledge of the other partitions. At most, the development of one partition should be aided by a set of guidelines for its applicability to the target TSP systems in general. The system integrator is responsible for guaranteeing a correct partition scheduling, so that partitions and the system as a whole meet their timing requisites. This may be done using schedulability analysis tools [55, 56].

3.4 Build and integration process

Development environments provide in general an adequate support for writing, building, debugging and maintaining computer-based applications, such as those present in desktop PCs as well as applications for embedded systems. This development process is known as the canonical build and integration process. It consists of using a compiler and/or assembler to build one or more object files that are linked together with a runtime library to form an executable image that's stored as a file on disk (on PCs) or in ROM memory in the case of embedded applications when no disk is available [57].

With respect to the development of software for partitioned architectures, such as those aboard aerospace systems, several limitations regarding the canonical build and integration process and standard build tools, such as those provided by GNU[1], need to be revisited and adapted to support the development of partitioned-based applications.

This way, because of the particularities of the AIR architecture, the software build and integration process differs from the canonical one, as provided by standard compilers and linkers. This process is pictured in Fig. 3.3 and Fig. 3.4, and it will now be described in detail.

3.4.1 Partition build process

The first stage concerns building each partition independently (Fig. 3.3). In the typical scenario, the applications to be executed in the context of a partition, the APEX library, and the underlying POS libraries (wrapped by the AIR PAL) may be provided by different teams or providers. Therefore, the build process is tailored to expect these independent object files, and link them together to produce an object file with no unresolved symbols but including relocation information (to allow linking with the remaining partitions). Although the AIR PAL also invokes the AIR PMK (which symbols are as of yet undefined), these interactions are wrapped using data structures to reference the appropriate primitives, which the AIR PMK will register by executing code generated at system integration time with the assistance of a specific AIR tool.

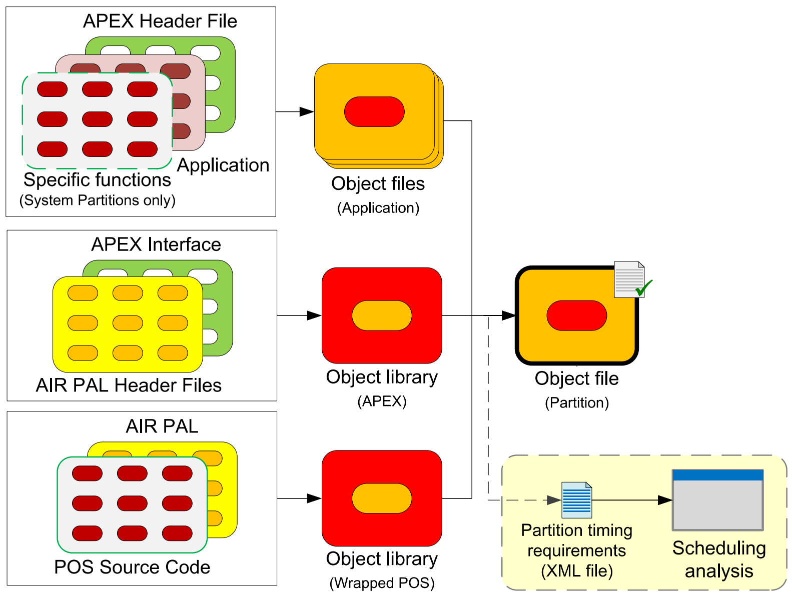


Figure 3.3: Software building by partition application developers

[1]http://www.gnu.org/

The introduction of a scheduling analysis phase in the application developers' software production chain [55] takes advantage of the composability properties, which will be defined next, to provide independent schedulability analysis. Application developers can perform this analysis using the timing requirements (period, worst-case execution time (WCET), deadline, etc.) of their applications' processes. This information can be either estimated, or tentatively determined through static code analysis [58, 59].

3.4.2 System integration

The system integration process (Fig. 3.4) receives input (partition object files) from potentially different teams or providers. Since all partitions will include the common interface provided by the AIR PAL and AIR APEX libraries, the various partitions' object files will have symbol name collisions; partitions running the same POS or POSs providing the same standardized interfaces (e. g., POSIX) have additional name collisions. Therefore, linking these objects will require previous preprocessing. This preprocessing can be in the form of a *tag filter* utility which prefixes all symbols and calls in each partition's object files with unique prefixes (e. g., `P1`, `P2`, etc.). This process can be further optimized by automating the generation of partition prefixes, namely deriving them from the configuration file.

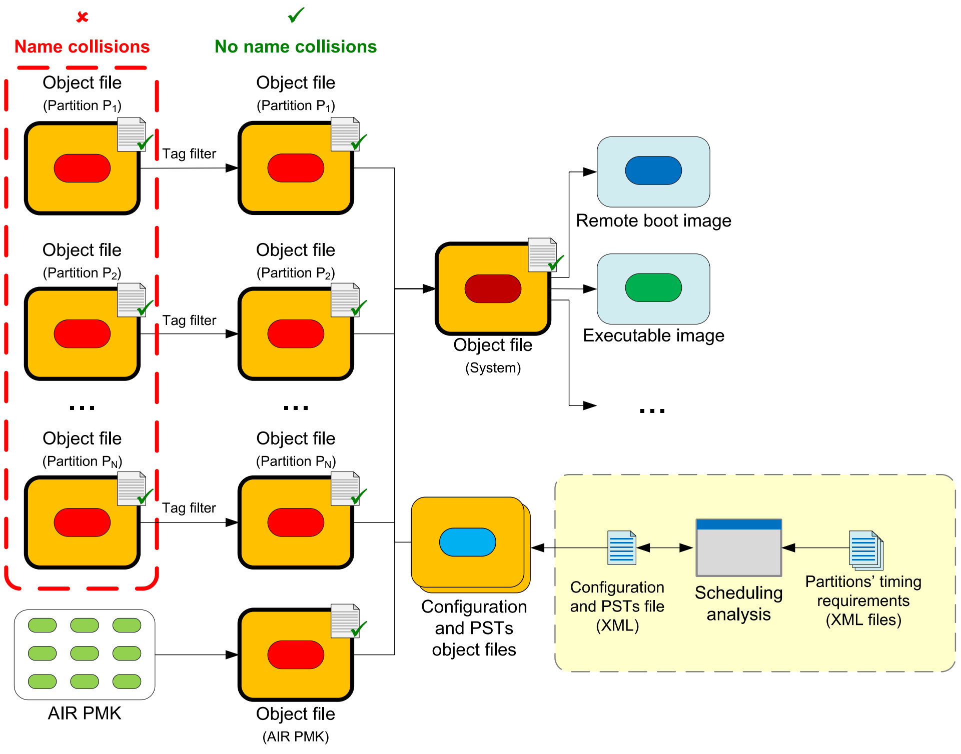


Figure 3.4: System integration

The partition objects can now be linked with the AIR PMK and the configuration object. This configuration object is derived by compiling C source code files, which in turn have been converted from XML (Extensible Markup Language) configuration files. The use of XML for the configuration file is motivated by the overall intention to comply, up to a certain degree, with the ARINC 653 specification [52]. Besides the parameters translated from these XML files (such as partition scheduling tables, addressing spaces, and interpartition communication ports and channels), configuration objects include routines for the AIR PMK to register the adequate primitives in the AIR PAL structures. This linking step produces the system object file, from which in turn one can generate the most adequate deployment format for the target platform. In the system integration phase, scheduling analysis capabilities shall be introduced in relation with the generation of a systemwide configuration [55].

3.4.3 Impact on the software development lifecycle

The applications development process for partitioned architectures modifies the classic software lifecycle, usually referred to as the "V" model, since it allows the development of each partition independently by different teams or providers. The impact over the classic software lifecycle, represented in Fig. 3.5, is observed in the stages performed independently: partition requirements, design, implementation, and application testing. Figure 3.5 represents the impact resulting from the independent development of different partitions (Fig. 3.3) and the system integration process (Fig. 3.4). The system validation is performed facing the mission requirements through the system testing [60].

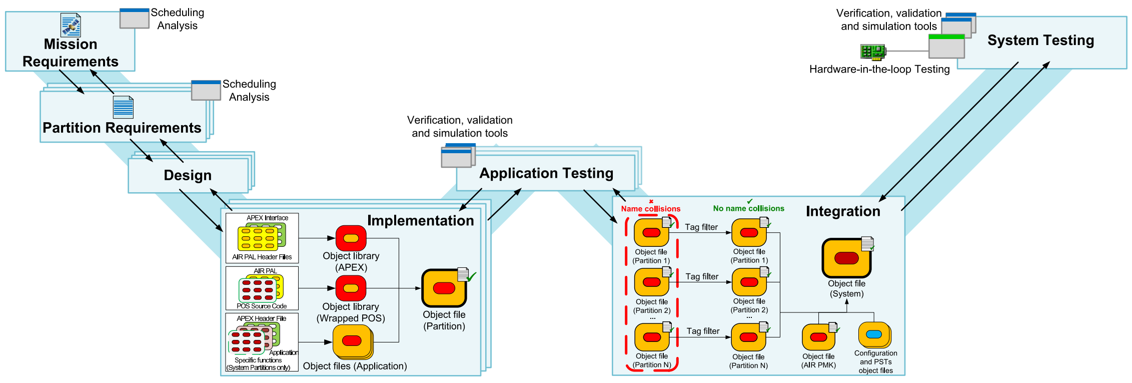


Figure 3.5: Impact on the software development lifecycle

3.5 Summary

This chapter described the AIR architecture, detailing the components with relevance to the development of this thesis. It explained the AIR schedulability and composability, followed by a complete description of the AIR build and integration process.

The following chapter will focus on the problem definition, which involves the description of the computational model adopted and the open challenges concerning the update of partition scheduling tables and software components.

Chapter 4

Problem definition

This chapter does a complete presentation of the issues regarding onboard update in the context of partitioned architectures for aerospace systems. It starts by characterizing the computational model used in the definition of a partitioned environment. It highlights the assumptions that will be taken to approach the problems discussed in the development of this thesis.

4.1 Computational model

Beyond embedded software general requirements for reliability, performance and cost, as well as the specific TSP requisites, there are several aspects that deserve some attention. Regarding the computational system and the AIR architecture, these aspects need to be specified adequately in order to know the requirements and limits of the solutions approached to improve the AIR architecture with the support for onboard software update.

4.1.1 Defining requirements and components

The focus of this work centers on the procedures taken in the spacecraft computational platform after receiving the update data from the communication subsystem. It is assumed that the upload of modified software components and partition scheduling tables (PSTs) is supported by a (secure) communication channel and data communication protocol. The data sent by the ground station, which may consist of a new set of PSTs or partition application components in the scope of onboard updates, are received by the system partition associated to communication functions. This partition is responsible for (i) the identification of the components to be updated; (ii) the allocation of the required memory resources, and; (iii) the functional integration of each component. It is assigned a guaranteed processing time to this partition in order to execute the referred operations.

It is assumed that the modified components have been subjected to offline V & V and were submitted to a set of tests in order to ensure that its integration on the target system will not change the system's correct behaviour or affect its safety.

To support the introduction of onboard software update operations, the original APEX interface must be extended to cope with new services, provided by an extended APEX (XAPEX) interface. These services are related to the update of PSTs sets and partition software components, and would be addressed in the next chapters. This XAPEX interface will be provided only to specific partitions, such as the partition that will perform the update procedures.

4.1.2 Integration on spacecraft onboard platform

The several spacecraft functions will be executed in independent partitions. Some of these functions are related to the system operation (avionics) whereas others operate the payload itself. The system partitions are those that usually must not stop their execution since it may be vital to the survivability of the spacecraft. Although, there may be scenarios where they can be momentarily stopped, such in the case of the AOCS functions when the space vehicle is stabilized in orbit. This small down-time may be used to perform update operations. It is noteworthy that it could also happen scenarios where stopping application partitions (i. e., the spacecraft payload) could bring undesired consequences and high costs. For example, stopping the payload functions of a telecommunications satellite will not result in any damage to the space vehicle but will shut down TV, Internet or telephone connections for thousands of clients, implying high monetary losses to the telecommunications provider.

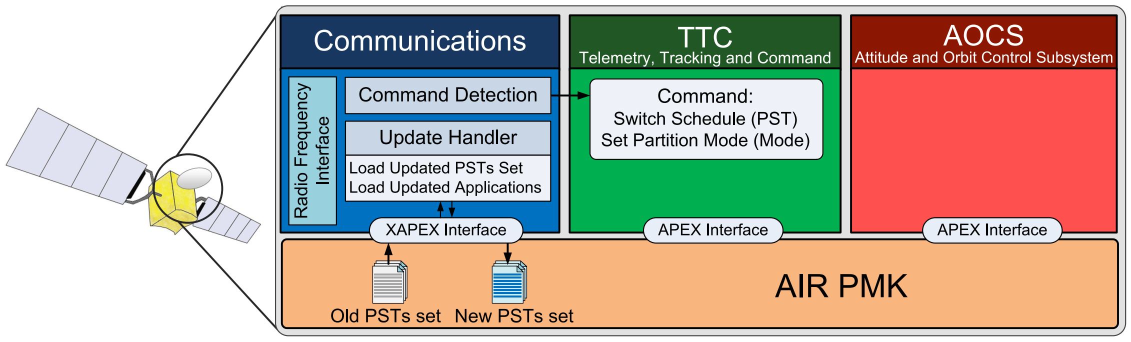


Figure 4.1: Spacecraft time- and space-partitioned computing platform

The entity responsible for controlling the update of partition applications and PSTs is the *Update Handler*, which will be hosted in the partition associated to the communications. The Update Handler, defined as a process/thread, interacts with the AIR PMK through calls to XAPEX primitives to perform the updates. The communications partition includes also a component for command detection, which passes the commands issued from the ground mission control to the TTC subsystem through an interpartition communication channel. Examples include a ground command to switch schedule or to set a partition's mode. Figure 4.1 illustrates a spacecraft computational system hosting communications, TTC and AOCS partitions, highlighting the operations concerning the loading of PSTs.

In AIR architecture, the partition responsible for the communications subsystem assumes executive functions assigned by the TTC, concerning the process of transfer of updated components and configurations. This avoids to exchange large amounts of information through communication channels between partitions thus increasing the efficiency of these operations. Memory management functions are also delegated to the communications subsystem by the TTC for the same reason.

4.2 Open challenges in onboard software update

The challenges that come to allow the inclusion of new features on a spacecraft during a mission are related to maintaining the real-time and safety guarantees of the original mission [13]. This should not affect the correct overall behaviour of the system, including the timeliness of the already running applications, and requires the verification and validation of the software components. This way, there are mission/flight phases that imply a best-effort approach to perform the update operations. This could involve switch to a scheduling mode where a shorter execution time is assigned to the communications partition running the Update Handler. This could also be achieved through tasks priority adjustments inside the communications partition, without the need to switch schedule. The space vehicle may be out of range of a ground station during the updates, which suggests that the Update Handler must also be prepared for the occurrence of communication blackouts and to achieve the expected results without affecting the correct system behaviour, in an autonomous way, i. e., without the need to interact with the ground mission control.

4.2.1 Problem definition

This section addresses two distinct scenarios, which one corresponding to a problem with different requisites. The first one concerns updating a set of PSTs, whereas the second is related to the update of application software components hosted in partitions. Actually, a third scenario may exist involving the occurrence of the previous two scenarios simultaneously. This consists of the issue by the ground mission control of a single update message containing either the set of modified PSTs and the modified software application components.

Updating a set of partition scheduling tables

The update of a set of PSTs consists of making available to the partition scheduler a modified set of PSTs, in order to meet new temporal requirements, such as those imposed by severe unexpected events or drastic changes in the environmental conditions. This operation involves providing the new set of PSTs to the Partition Scheduler component of the AIR PMK. The activation of the new set of PSTs must guarantee the safety of the switch operation between the old and the new sets of PSTs, thus ensuring that the correctness of partition scheduling is not compromised.

Updating a partition in the idle mode

This scenario concerns the update of application software components hosted in a partition in the idle mode, being the method suitable for performing the update of partitions executing non-critical spacecraft functions, such as some payload functions. A partition is said to be in the idle mode when it is shut down and is not scheduling or executing any application processes [4, 52]. A partition may enter the idle mode after a command issued from the ground mission control, upon the detection of a partition internal fault, or explicitly by request of the partition's application. The update of software components that can be stopped is simplified when is possible to reuse the resources assigned to the old application to the updated one.

The update of a partition in the active mode, in which process scheduling is enable and application processes are being executed, has been left to future work.

4.2.2 Proposed methodology

The onboard update methodology that will be followed consists of the definition of a new set of PSTs and the modification of application software components, in order to upgrade the original mission adapting it according to new requirements.

The envisaged methodology can be divided in four main steps, that can be adapted for both the update of PST and the update of software application components. The first step is the offline verification and validation of software modifications and PSTs, corresponding to the AIR original verification and validation process to ensure that safety and timeliness would not be compromised with the introduction of new components. This first step is performed on the ground either by or at least under the coordination of the system integrator. The second step concerns the extraction of updated components, which consists of identifying the components that have been modified, extract them from the complete system object file and create a new one that will be uploaded to the spacecraft. This second step concludes the operations on the ground. The third step is the transfer of the received information, namely the updated PSTs and application components, to the appropriate memory addressing space in the spacecraft onboard computational infrastructure, after the reception of those elements by the spacecraft. The update of application components requires also to stop the old partition. The fourth and last step, also performed on the spacecraft, concerns the activation of updated components, which consists of the safe application of an updated set of PSTs, or the proper placement and activation of updated software components.

The further development of these steps and the practical implementation details, along with the relevant results, will be addressed in the next two chapters. The modifications introduced by this methodology to the AIR native build and integration process will also be addressed.

4.3 Summary

This chapter presented the computational model assumed for the development of onboard update solutions. It described the open challenges in onboard update and defined the problem approached in this thesis, introducing also the core ideas for the methodology which will be addressed in detail in the following chapters.

The contributions of this chapter produced the following conference paper:

J. Rosa, J. Craveiro and J. Rufino, "Exploiting AIR Composability towards Spacecraft Onboard Software Update", in *Actas do INForum 2010, Simpósio de Informática*, Braga, Portugal, Sep. 2010. [13]

The next chapter describes the methodology, algorithms and implementation results for the update of partition scheduling tables.

Chapter 5

Updating partition schedules

This chapter addresses the update of partition scheduling tables in time- and space-partitioned systems. The methodology defined here was motivated by spaceborne systems' need to adapt to changing conditions and unexpected events. This chapter details the AIR partition scheduling and its relation with reconfigurability. Then, it is described and analysed the algorithm for updating PSTs. This chapter ends with the presentation of the proof-of-concept prototype and results.

5.1 Scheduling partitions

As was introduced in Chapter 3, the AIR architecture uses a two-level scheduling scheme (remember Fig. 3.2) and incorporates mode-based partition schedules. The support for mode-based schedules is provided through APEX services. The APEX_SWITCH_SCHEDULE service, represented in Algorithm 1, sets the schedule that will start executing at the begin of the next major time frame (MTF). This primitive receives, as its only parameter, the $scheduleId$, referencing the index of the next PST to be used.

Algorithm 1 APEX_SWITCH_SCHEDULE primitive

```
1: function APEX_SWITCH_SCHEDULE($scheduleId$)
2:     $nextSchedule \leftarrow scheduleId$
3: end function
```

The AIR Partition Scheduler, responsible for guaranteeing to make a schedule switch effective at the end of the respective MTF, is 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 happens only after the end of the current MTF, in order to maintain the timeliness, this may result on a waiting time before the PSTs switching [4]. 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) [4].

Algorithm 2 AIR Partition Scheduler (mode-based schedules)

```
 1: $ticks \leftarrow ticks + 1$                    ▷ $ticks$: global system clock tick counter
 2: if $schedules_{currentSchedule}.table_{tableIterator}.tick = (ticks - lastScheduleSwitch) \bmod schedules_{currentSchedule}.mtf$ then
 3:     if $currentSchedule \neq nextSchedule \wedge (ticks - lastScheduleSwitch) \bmod schedules_{currentSchedule}.mtf = 0$ then
 4:         $currentSchedule \leftarrow nextSchedule$
 5:         $lastScheduleSwitch \leftarrow ticks$
 6:         $tableIterator \leftarrow 0$
 7:     end if
 8:     $heirPartition \leftarrow schedules_{currentSchedule}.table_{tableIterator}.partition$
 9:     $tableIterator \leftarrow (tableIterator + 1) \bmod schedules_{currentSchedule}.numberPartitionWindows$
10: end if
```

5.2 Safe update of partition schedules

5.2.1 The XAPEX_PSTUPDATE service

To support the introduction of the operation for update PSTs, the original APEX interface was extended with an appropriate service. The XAPEX_PSTUPDATE 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. 4.1 (Chapter 4, page 30). Algorithm 3 concerns the pseudo-code representation of the XAPEX_PSTUPDATE service.

Algorithm 3 XAPEX_PSTUPDATE primitive

```
1:  function XAPEX_PSTUPDATE($newSchedules$)
2:      while $\neg safePstUpdate$ do
3:          if $currentSchedule = nextSchedule$ then
4:              for $newSchedules_i \in newSchedules$ do
5:                  if $newSchedules_i \equiv schedules_{currentSchedule}$ then
6:                      $safePstUpdate \leftarrow$ TRUE
7:                      $newCurrentSchedule \leftarrow i$
8:                      break
9:                  end if
10:             end for
11:         end if
12:     end while
13:     SWAP($schedules$,$newSchedules$)
14:     $currentSchedule \leftarrow newCurrentSchedule$
15: end function
```

5.2.2 Onboard update of PSTs

The methodology for the onboard update of PSTs consists of the definition of a new set of PSTs to reconfigure the mission according to new requirements. The activation of a new set of PSTs must guarantee the safety of switch between the old and the new sets of PSTs, thus ensuring that the correctness of system scheduling is not compromised. This is illustrated in Fig. 5.1, which will be clarified later in the algorithm description. The update methodology consists of a four-step procedure described as follows.

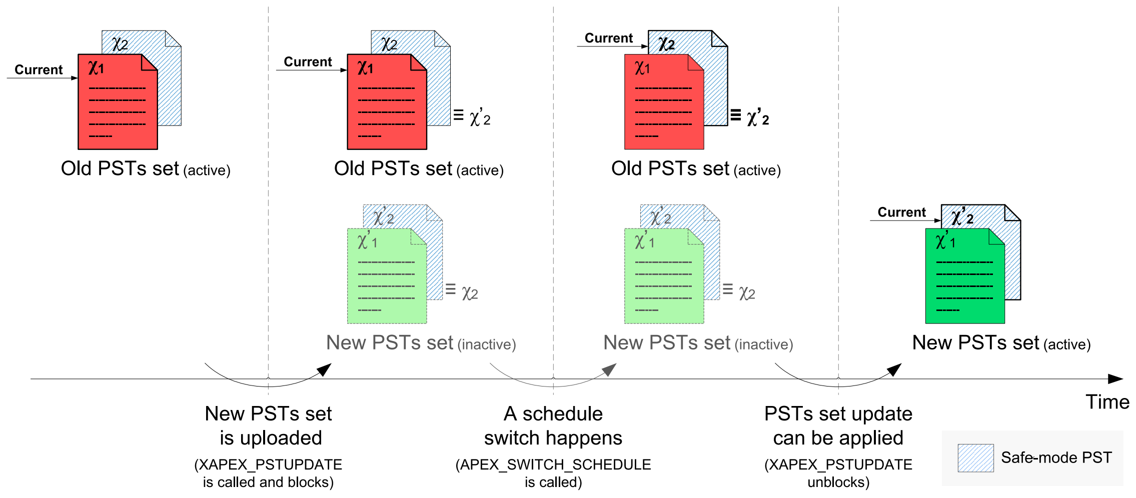


Figure 5.1: Update of a set of PSTs

Offline verification and validation of redefined PSTs

The definition of new sets of PSTs must involve the verification and validation of the updated 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 [55, 56]. This step benefits from AIR composability which allows the verification and validation of PSTs to be done by software development teams or providers independently.

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 PSTs set will be uploaded to the spacecraft onboard computer.

Transfer of redefined PSTs

In the spacecraft, the PSTs are received by the partition hosting the communication functions (Fig. 4.1). 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.

Activation of redefined PSTs

The first condition for 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 only become effective at the end of the current MTF. 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 4–7). 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. 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. 5.1. To allow 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. 5.1 as χ_2 and χ_2').

5.2.3 Achieving reconfigurability

System reconfiguration facing new mission constraints is essential to ensure the survivability of the spacecraft and the mission itself. The support for robust reconfiguration in the AIR architecture is done through mechanisms such as mode-based schedules; health monitor, responsible for handling and containing errors to their domains of occurrence; process deadline violation monitoring, to detect deadline violations of the partitions' timing requirements, and; low-level event overload control, to control the timeliness of asynchronous events [61, 4]. These mechanisms enable the safe reconfiguration of system components. By offering the possibility to host natively multiple PSTs and switch among them on demand during the execution of the system, AIR allows for (self-)adaptation of the system to the mission's different phases and to operational condition changes [61]. 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 AOCS when an event implies changing the partitions' temporal requirements. Furthermore, the inclusion of onboard update of PSTs features in the AIR architecture introduces another level of reconfigurability since it allows the introduction of new PSTs according to new requirements imposed by unexpected events during a mission.

5.3 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.

5.3.1 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 [62]. 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 5.1 shows the logical SLOC and CC values for the C implementation of the XAPEX_PSTUPDATE primitive and the AIR Partition Scheduler [61].

Table 5.1: Logical SLOC and cyclomatic complexity (CC) for the XAPEX_PST-UPDATE primitive and the AIR Partition Scheduler

	Logical SLOC	**CC**
XAPEX_PSTUPDATE	9	4
AIR Partition Scheduler	13	4

5.3.2 Computational complexity

This section focus on the computational complexity analysis for the XAPEX_PSTUPDATE primitive (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.

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 it may be necessary to compare all the PSTs in the updated set until 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. 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.

5.4 Proof-of-concept prototype and evaluation

5.4.1 Prototyping

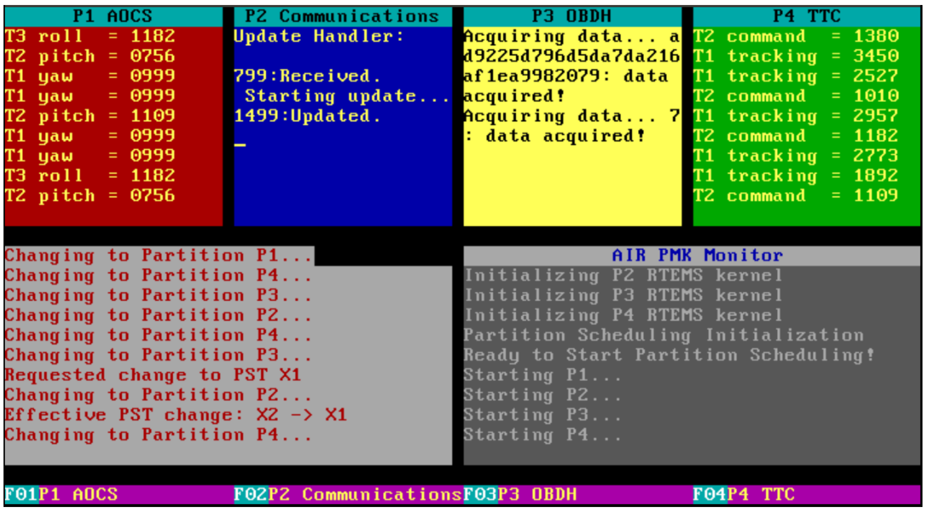
Aiming to demonstrate the onboard update of partition scheduling tables, it was modified an existing prototype of an AIR-based system to include the facilities for the update of PSTs.

The AIR prototype is constituted by several mockup applications based on

Real-Time Executive for Multiprocessor Systems (RTEMS), version 4.8.1 [63].

The prototype includes four partitions, each one running an application representing typical spacecraft functions (a subset of these functions are pictured in Fig. 5.2). 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 [64], illustrated in Fig. 5.2. 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 [65].



```
P1 AOCS
T3 roll   = 1182
T2 pitch = 0756
T1 yaw    = 0999
T1 yaw    = 0999
T2 pitch = 1109
T1 yaw    = 0999
T1 yaw    = 0999
T3 roll   = 1182
T2 pitch = 0756
```

```
P2 Communications
Update Handler:

799:Received.
 Starting update...
1499:Updated.
_
```

```
P3 OBDH
Acquiring data... a
d9225d796d5da7da216
af1ea9982079: data
acquired!
Acquiring data... 7
: data acquired!
```

```
P4 TTC
T2 command   = 1380
T1 tracking = 3450
T1 tracking = 2527
T2 command   = 1010
T1 tracking = 2957
T2 command   = 1182
T1 tracking = 2773
T1 tracking = 1892
T2 command   = 1109
```

```
Changing to Partition P1...
Changing to Partition P4...
Changing to Partition P3...
Changing to Partition P2...
Changing to Partition P4...
Changing to Partition P3...
Requested change to PST X1
Changing to Partition P2...
Effective PST change: X2 -> X1
Changing to Partition P4...
```

```
AIR PMK Monitor
Initializing P2 RTEMS kernel
Initializing P3 RTEMS kernel
Initializing P4 RTEMS kernel
Partition Scheduling Initialization
Ready to Start Partition Scheduling!
Starting P1...
Starting P2...
Starting P3...
Starting P4...
```

```
F01P1 AOCS          F02P2 CommunicationsF03P3 OBDH          F04P4 TTC
```

Figure 5.2: Prototype implementation demonstration, featuring the VITRAL text-mode windows manager for RTEMS

5.4.2 Evaluation

The behaviour of the partition scheduling was analysed during the update of a new set of PSTs. In order to test different scenarios and compare the results, the demonstration was set up with different possible configurations. Then, there were performed 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), and; the request to switch to a different schedule, through a call to the APEX_SWITCH_SCHEDULE primitive (Algorithm 1). 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 [4].

Table 5.2: Partition scheduling tables used in test scenarios (for the prototype implementation demonstration)

Preemption point	Partitions (per PST)		
(time units)	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

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. 5.3 and Table 5.2. 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. 5.1). The referred PSTs have all a MTF of 1300 time units.

At first were defined four base scenarios and then were discussed 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. 5.1.

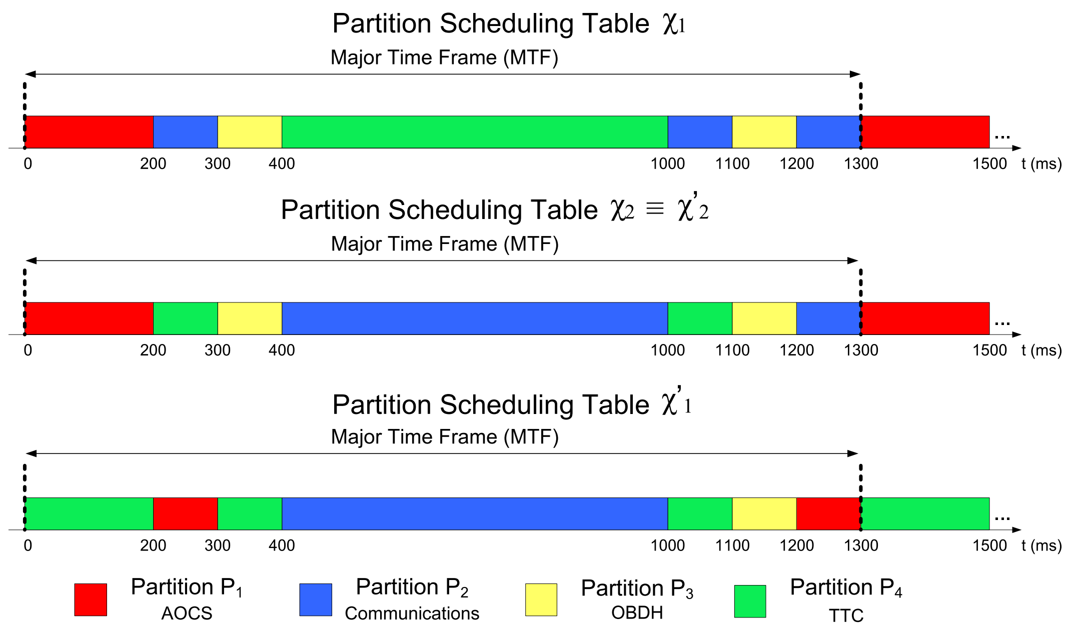


Figure 5.3: Partition scheduling tables used in test scenarios

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 the 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 a 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 test scenarios 1 to 4.

5.5 Summary

This chapter 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. This chapter detailed and discussed the algorithm behind the onboard update methodology.

The main results of this chapter produced two publications, listed below. The reconfigurability issues addressed in Section 5.2.3 were published in:

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. [12]

A second article, accepted for publication at the time of the writing of this thesis, described the technical details, demonstration prototype, and evaluation of the methodology for update of partition scheduling tables:

J. Rosa, J. Craveiro, and J. Rufino, “Safe Online Reconfiguration of Time- and Space-Partitioned Systems”, in *9th IEEE International Conference on Industrial Informatics (INDIN'2011)*, Caparica, Lisboa, Portugal, Jul. 2011, accepted for publication. [11]

The next chapter describes the methodology, algorithms and implementation results for the update of partition software components.

Chapter 6

Updating partition software components

This chapter addresses the update of partitions in time- and space-partitioned systems. Here, the focus is on the update of partitions on idle mode. A partition in the idle mode is shut down and not scheduling or executing application processes [4, 52]. This chapter describes the several steps of a methodology for update partitions. Then, it explains how adaptability can be achieved with the possibility to perform update of partitions, and the impact of the update methodology on the software development process. Finally, this chapter does the presentation of the proof-of-concept prototype and evaluation.

6.1 Safe update of partitions

6.1.1 The XAPEX_PUPDATE service

Similarly to the creation of the primitive XAPEX_PSTUPDATE to enable the updates of PSTs, the APEX need to be extended with a specific service, named XAPEX_PUPDATE, to allow the update of partitions. This service is provided in the XAPEX interface to specifically authorized partitions, such as the communications partition. The XAPEX_PUPDATE primitive concerns the update of partitions on the idle mode and its main goal is the safe replace of an old partition by a new one with updated software components.

The primitive XAPEX_PUPDATE receives the *id* of the partition to be updated and the updated software components as specified in Algorithm 4. This primitive simply invokes an AIR PMK function that will initiate the update process (Algorithm 4). The execution control of the partition under update is transferred to the AIR PMK during this process. For simplification purposes, the Update Handler provides the partition and the updated software components on the same call to the AIR PMK.

Algorithm 4 XAPEX_PUPDATE primitive

```
1: function XAPEX_PUPDATE($partitionId$,$updatedComponents$)
2:     PMK_PUPDATE($partitionId$,$updatedComponents$)
3: end function
```

It is foreseen the inclusion of dynamic memory allocation/deallocation services, presented in Table 6.1, in the XAPEX interface in order to enable the update of partitions in the active mode, which is the normal operational mode of a partition, i. e., in which processes are being executed [4, 52]. The original APEX interface does not include dynamic memory allocation/deallocation services due to the non-determinism that these operations could introduce to the system execution thus affecting the timeliness of operations. The adoption of these services in real-time systems requires a deep analysis and may involve the study of the recommendations present in standards for the implementation of embedded software, such as MISRA C rules [42] or Galileo SW-DAL B [66]. In the context of this thesis' work, an early implementation of the referred services has been approached, although this needs to be consolidated in the future.

Regarding memory management aspects, the methodology described in this thesis assumes that the new version of the partition that is being updated does not exceed the memory size of the partition that will be replaced, this is, the updated partition can be placed in the same memory addressing space of the partition being updated without the use of dynamic memory allocation/deallocation techniques.

Table 6.1: XAPEX memory management services foreseen

Primitive	**Short description**
XAPEX_MALLOC	Allocate memory from the partition's free memory pool
XAPEX_MFREE	Deallocate a memory zone for the partition's free memory pool
XAPEX_MCLAIM	Claim memory from a specified partition for the partition's free memory pool

The step-by-step methodology implemented for the update of partitions in the idle mode is described as follows.

6.1.2 Onboard update of partitions in the idle mode

The update procedures of a partition whose execution of software components can be stopped are simplified to a certain extent, allowing claim in advance and reuse the memory of the original application to host the updated software components of the partition. After the update of partition software components the application should be restarted using a cold start procedure, which means it will be restarted discarding the previous execution context.

Offline verification and validation of software modifications

This step corresponds to the AIR original verification and validation process of software components to ensure that safety and timeliness would not be compromised with the introduction of new components. Due to the composability properties, this may be done by software development teams or providers independently. Application testing benefits from the inclusion of tools such IMADE, from the GMVIS-Skysoft Portugal company, which supports the development, test, simulation and analysis of software applications without the need to access the target platform [67].

Extraction of updated components

The final goal of this step is to identify which components need to be uploaded to the spacecraft onboard computer, extract them from the complete system object file, and create a new one composed only of the modified components. Using appropriate tools, this object file should be built according to a specific format in order to be recognized by the Update Handler. After this step, the new object file will be uploaded to the spacecraft.

Transfer of the updated components

In the spacecraft, the application software components are received by the Update Handler in the partition hosting the communication functions. The Update Handler upon reception of these components invokes the XAPEX_PUPDATE primitive to initiate the update process.

The update of a partition requires stopping that partition, i. e., change it to the idle mode, to do the replacement. A first approach involves the modification of the original *AIR PMK Time Manager* internal component, responsible for handle the scheduling and dispatching of partitions, in order to stop the dispatch of the partition that is being updated. Then, the partition will stop its execution calling the appropriate APEX primitive.

The PMK Time Manager, implemented by the *AIR_clock_tick* function represented in Algorithm 5, passes the interrupt to the PMK Partition Scheduler and then to the *PMK Partition Dispatcher*, responsible for secure the management of all provisions required to guarantee spatial segregation [4]. The *AIR_clock_tick* function is called at every system clock tick interrupts, passed to the PMK Time Manager. This function, presented in Algorithm 6, was modified in order to jump the partition dispatcher during an update. This happens when the condition $updateActivated \wedge \_heirPartition.number = partitionId$ is true (Algorithm 6, line 4). This allows the AIR PMK to perform the update. The scheduler, however, continues its normal execution. At this level, the update is completely transparent.

Algorithm 5 AIR PMK Time Manager (original)

```
1: function AIR_CLOCK_TICK
2:     $number\_total\_clock\_ticks \leftarrow number\_total\_clock\_ticks + 1$
3:     AIR_PARTITION_SCHEDULER( )
4:     AIR_PARTITION_DISPATCHER( )
5:     if $hasInitialized$ then
6:         $number\_active\_clock\_ticks \leftarrow number\_active\_clock\_ticks + 1$
7:     end if
8: end function
```

Algorithm 6 Modified version of the AIR PMK Time Manager

```
1: function AIR_CLOCK_TICK
2:     $number\_total\_clock\_ticks \leftarrow number\_total\_clock\_ticks + 1$
3:     AIR_PARTITION_SCHEDULER( )
4:     if $\neg(updateActivated \wedge \_heirPartition.number = partitionId)$ then
5:         AIR_PARTITION_DISPATCHER( )
6:     end if
7:     if $hasInitialized$ then
8:         $number\_active\_clock\_ticks \leftarrow number\_active\_clock\_ticks + 1$
9:     end if
10: end function
```

The same result could be achieved modifying the current partition scheduling, using a specific PST to perform the updates. However, this would imply to have available specific PSTs to permit the update of any partition required (i. e., a PST without the partition P_1 to allow the update of P_1, other without P_2, and so on), and performing schedule switches between PSTs (a schedule switch to the PST without the partition being updated, and a switch to the original PST after the update). This approach, although, involves the execution of several operations, whereas the technique described in Algorithm 6 simply lies in the validation of a condition (Algorithm 6, line 4), at a lower level.

The AIR PMK, after stopping the dispatch of the partition being updated upon the call from XAPEX_PUPDATE, will signal that partition to finalize its execution. The partition, duly authorized to this effect, performs its shut down invoking an APEX primitive, which in turn calls the appropriate POS function.

Activation of updated components

As soon as the old partition ends its execution through the shut down process, the AIR PMK proceeds with the proper placement of the updated software components in the memory addressing space of the partition. The approach followed is simplified to facilitate the integration of the updated components in the memory space assigned to the old partition thereby reusing and sparing memory resources.

The new version of the partition is then initialized through a cold start action discarding the execution context of the old partition. This initialization is done through a process similar to the one performed during the initialization of the partition operating systems at system startup.

6.1.3 Achieving adaptability

Adaptation to changing or unexpected conditions is of great importance for a mission's survival. AIR architecture employs mechanisms such as mode-based schedules to maintain or improve the system effectiveness when facing internal or external changes [61]. Furthermore, the possibility of update partition software components, enabled through the methodology described in this thesis, adds more potential to the AIR architecture regarding system adaptability, since it allows the modification of partitions to include new features during the course of a mission. The methodology described offers flexible system adaptation as the partition software components updated are not forced or expected to have the functionalities provided by the old version of the partition.

6.1.4 Impact of the update methodology

The methodology defined for updating PSTs and software components introduces new constraints in the software development lifecycle, with major impact on the verification and validation phases, and introduces new steps, namely those related to the extraction of the modified components that will be transferred later to the spacecraft's onboard computing platform. The impact of the methodology is illustrated in Fig. 6.1 (cf. Fig. 3.5, Chapter 3, page 27), where only the partitions related to new mission requirements need to be modified, along with the new sets of PSTs. The model presented in Fig. 6.1 represents requirement analysis, design and implementation, followed by the extraction and format of the components that are being updated.

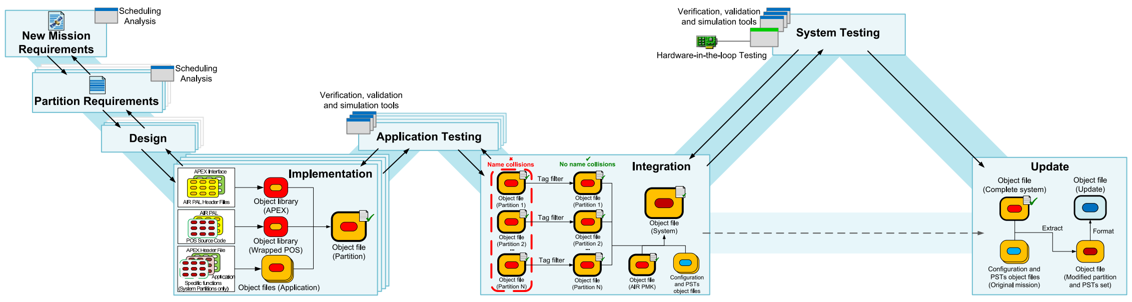

New Mission Requirements
Scheduling Analysis
Partition Requirements
Scheduling Analysis
Design
Verification, validation and simulation tools
Application Testing
Implementation
APEX Interface
AIR PAL Header Files
Object library (APEX)
AIR PAL
POS Source Code
Object library (Wrapped POS)
Object file (Partition)
APEX Header File
Application
Specific functions (System Partitions only)
Object files (Application)
Name collisions
No name collisions
Integration
Tag filter
Object file (Partition 1)
Object file (Partition 2)
Object file (Partition N)
Object file (System)
Object file (AIR PMK)
Configuration and PSTs object files
Verification, validation and simulation tools
Hardware-in-the-loop Testing
System Testing
Update
Object file (Complete system)
Object file (Update)
Extract
Format
Configuration and PSTs object files (Original mission)
Object file (Modified partition and PSTs set)

Figure 6.1: Impact of the update methodology on the software development life-cycle

6.2 Proof-of-concept prototype and evaluation

6.2.1 Prototyping

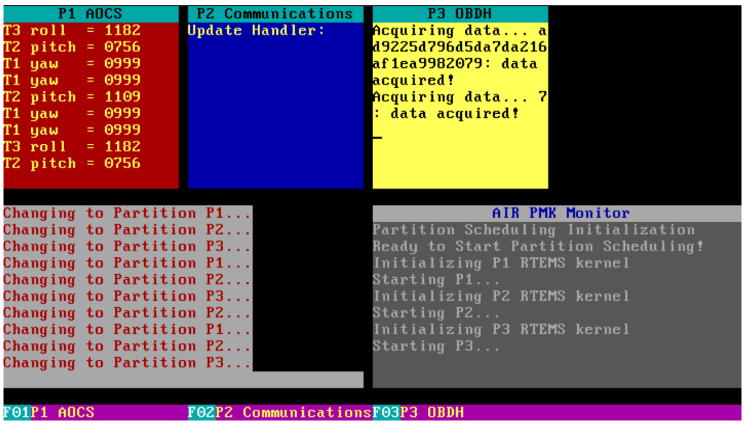
The functionality for the update of partitions was implemented in the scope of an AIR-based system prototype. For demonstration purposes, the prototype has three RTEMS-based mockup applications, each one concerning a spacecraft subsystem in the domain of a partition. Partition P_1 is associated to the AOCS; partition P_2 concerns the communications functions, being responsible for the reception and activation of the updates through the Update Handler component, and; partition P_3 features the OBDH subsystem. The partitions in this prototype are scheduled by the partition scheduling table χ_1, described in Table 6.2.

Table 6.2: Partition scheduling table χ_1 for the partitions update prototype

Preemption point (time units)	Partition
0	P_1
200	P_2
300	P_3
400	P_1
1000	P_2
1100	P_3
1200	P_2

MTF = 1300 time units

Like the prototype adapted for the demonstration of the safe update of PSTs, described in the previous chapter, the prototype concerning the update of partitions takes profit of VITRAL windows manager for RTEMS applications [64], illustrated in Fig. 6.2. VITRAL windows manager has an output window associated to each partition, a debug window illustrating the scheduling of partitions, and an AIR PMK Monitor window showing the initialization/termination of partitions. The activation of the update of partitions is made through interaction with keyboard, for demonstration purposes. This prototype was implemented for an Intel IA-32 target platform and tested on the QEMU emulator [65].

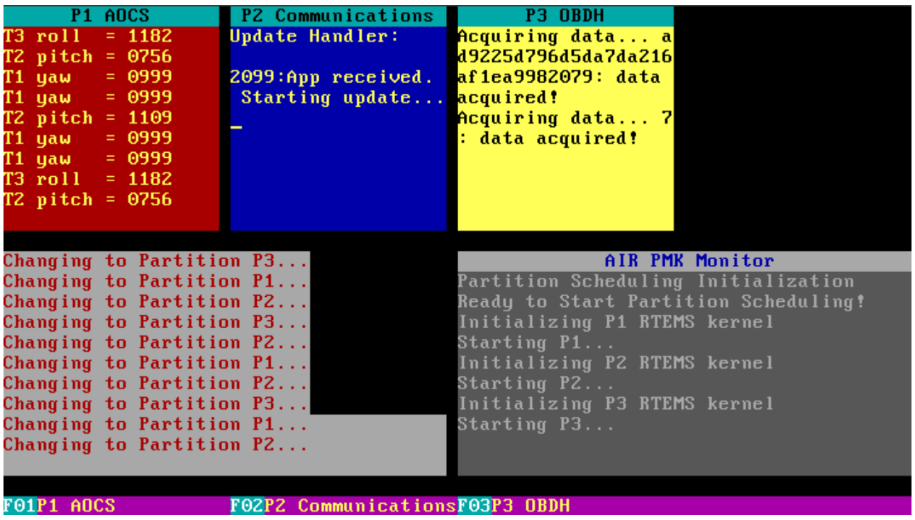


```
P1 AOCS            P2 Communications   P3 OBDH
T3 roll  = 1182    Update Handler:     Acquiring data... a
T2 pitch = 0756                        d9225d796d5da7da216
T1 yaw   = 0999                        af1ea9982079: data
T1 yaw   = 0999                        acquired!
T2 pitch = 1109                        Acquiring data... 7
T1 yaw   = 0999                        : data acquired!
T1 yaw   = 0999                        _
T3 roll  = 1182
T2 pitch = 0756

Changing to Partition P1...            AIR PMK Monitor
Changing to Partition P2...            Partition Scheduling Initialization
Changing to Partition P3...            Ready to Start Partition Scheduling!
Changing to Partition P1...            Initializing P1 RTEMS kernel
Changing to Partition P2...            Starting P1...
Changing to Partition P3...            Initializing P2 RTEMS kernel
Changing to Partition P2...            Starting P2...
Changing to Partition P1...            Initializing P3 RTEMS kernel
Changing to Partition P2...            Starting P3...
Changing to Partition P3...

F01P1 AOCS         F02P2 CommunicationsF03P3 OBDH
```

Figure 6.2: Prototype for the update of partitions, featuring the VITRAL text-mode windows manager for RTEMS

6.2.2 Evaluation

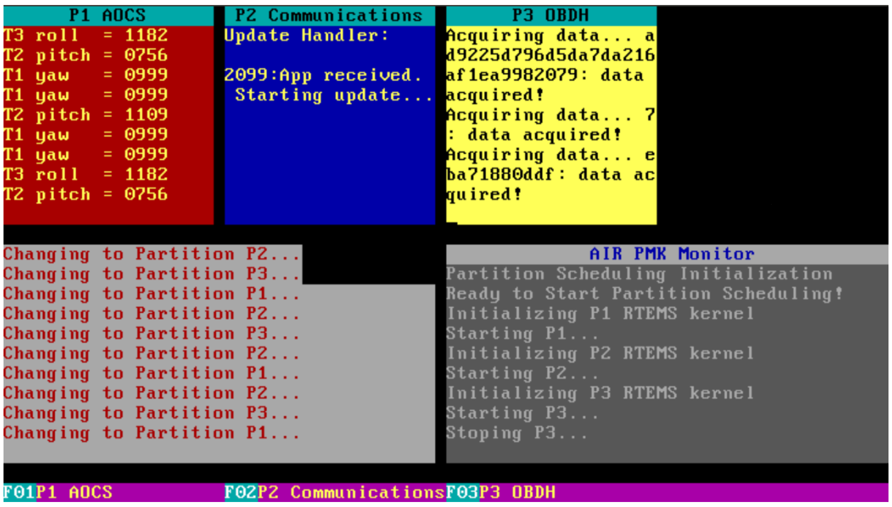
The update of a partition was tested over the partition P_3, simulating OBDH functions. The activation of the update, illustrated in the communications window (partition P_2 running the Update Handler), Fig. 6.3, triggers the update process at instant 2099, according to the system clock.



```
P1 AOCS            P2 Communications   P3 OBDH
T3 roll  = 1182    Update Handler:     Acquiring data... a
T2 pitch = 0756                        d9225d796d5da7da216
T1 yaw   = 0999    2099:App received.  af1ea9982079: data
T1 yaw   = 0999     Starting update... acquired!
T2 pitch = 1109    _                   Acquiring data... 7
T1 yaw   = 0999                        : data acquired!
T1 yaw   = 0999
T3 roll  = 1182
T2 pitch = 0756

Changing to Partition P3...            AIR PMK Monitor
Changing to Partition P1...            Partition Scheduling Initialization
Changing to Partition P2...            Ready to Start Partition Scheduling!
Changing to Partition P3...            Initializing P1 RTEMS kernel
Changing to Partition P2...            Starting P1...
Changing to Partition P1...            Initializing P2 RTEMS kernel
Changing to Partition P2...            Starting P2...
Changing to Partition P3...            Initializing P3 RTEMS kernel
Changing to Partition P1...            Starting P3...
Changing to Partition P2...

F01P1 AOCS         F02P2 CommunicationsF03P3 OBDH
```

Figure 6.3: Activation of the partitions update operation

Figure 6.4 illustrates the moment after the finalization of the execution of P_3, described in the update methodology. This can be observed in the last output line of the AIR PMK Monitor window. Finally, Fig. 6.5 illustrates the moment after the update process. It shows, in the communications window, that the update has ended at instant 3399. The AIR PMK Monitor window shows the initialization of the update partition, referred to as P_3', executing the new version of the OBDH subsystem. P_3' is a modified version of the partition P_3 with different text labels, for demonstration purposes. The only differences between the two versions of the OBDH partition concern the strings in the source code. The original partition P_3 had the labels "Acquiring data..." and ": data acquired!" whereas the new partition P_3' has the labels "Handling data..." and ": data handled!". The modifications resulted from the update can easily be observed comparing the P_3 OBDH window in Fig. 6.4, for instance, with its updated version P_3' in Fig. 6.5.



```
P1 AOCS
T3 roll  = 1182
T2 pitch = 0756
T1 yaw   = 0999
T1 yaw   = 0999
T2 pitch = 1109
T1 yaw   = 0999
T1 yaw   = 0999
T3 roll  = 1182
T2 pitch = 0756
```

```
P2 Communications
Update Handler:

2099:App received.
 Starting update...
```

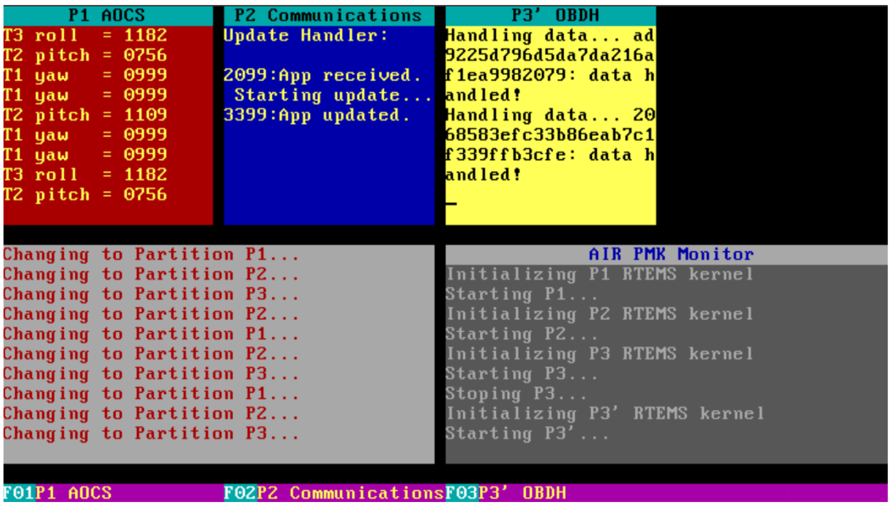
```
P3 OBDH
Acquiring data... a
d9225d796d5da7da216
af1ea9982079: data
acquired!
Acquiring data... 7
: data acquired!
Acquiring data... e
ba71880ddf: data ac
quired!
_
```

```
Changing to Partition P2...
Changing to Partition P3...
Changing to Partition P1...
Changing to Partition P2...
Changing to Partition P3...
Changing to Partition P2...
Changing to Partition P1...
Changing to Partition P2...
Changing to Partition P3...
Changing to Partition P1...
```

```
AIR PMK Monitor
Partition Scheduling Initialization
Ready to Start Partition Scheduling!
Initializing P1 RTEMS kernel
Starting P1...
Initializing P2 RTEMS kernel
Starting P2...
Initializing P3 RTEMS kernel
Starting P3...
Stoping P3...
```

```
F01P1 AOCS          F02P2 CommunicationsF03P3 OBDH
```

Figure 6.4: Original partition P_3 has stopped its execution



```
P1 AOCS
T3 roll  = 1182
T2 pitch = 0756
T1 yaw   = 0999
T1 yaw   = 0999
T2 pitch = 1109
T1 yaw   = 0999
T1 yaw   = 0999
T3 roll  = 1182
T2 pitch = 0756
```

```
P2 Communications
Update Handler:

2099:App received.
 Starting update...
3399:App updated.
```

```
P3' OBDH
Handling data... ad
9225d796d5da7da216a
f1ea9982079: data h
andled!
Handling data... 20
68583efc33b86eab7c1
f339ffb3cfe: data h
andled!

_
```

```
Changing to Partition P1...
Changing to Partition P2...
Changing to Partition P3...
Changing to Partition P2...
Changing to Partition P1...
Changing to Partition P2...
Changing to Partition P3...
Changing to Partition P1...
Changing to Partition P2...
Changing to Partition P3...
```

```
AIR PMK Monitor
Initializing P1 RTEMS kernel
Starting P1...
Initializing P2 RTEMS kernel
Starting P2...
Initializing P3 RTEMS kernel
Starting P3...
Stoping P3...
Initializing P3' RTEMS kernel
Starting P3'...
```

```
F01P1 AOCS          F02P2 CommunicationsF03P3' OBDH
```

Figure 6.5: The updated partition P_3' has replaced the original partition P_3 and was initialized

6.3 Summary

This chapter addressed a methodology for the update of partition software components. It focused on the update of partitions in the idle mode. It described the XAPEX_PUPDATE service, to allow the activation of the update. Then, this chapter addressed adaptability concerns and explained the impact of the update methodology on the software development process. Finally, this chapter presented the proof-of-concept prototype and demonstration of the integration of partitions update features on the AIR architecture.

The adaptability concerns, addressed in Section 6.1.3, contributed the results presented in the following publication:

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. [12]

The next chapter ends this dissertation, issuing concluding remarks and future work.

Chapter 7

Conclusion

The objective of this thesis was the development of an onboard reconfiguration and update methodology for time- and space-partitioned (TSP) architectures aiming the aerospace systems, having the AIR architecture, an ARINC 653-based partitioned architecture for aerospace applications featuring strong temporal and spatial segregation, as the background technology.

The development of the onboard reconfiguration and update methodology was motivated by the need to adapt to changing and unexpected environmental conditions, thus overcome severe incidents or internal failures. This way, it is of extreme importance to have the possibility to reconfigure system parameters, such as partition scheduling tables, and to update partition software components during a mission, i. e., during the operational mode of a space vehicle and without the need to stop the system execution. This may contribute to reach a safe system upon the occurrence of environmental changes or spacecraft failures, thus increasing the mission's survivability.

The main contributions of this work were: (i) the definition of an onboard update methodology for TSP architectures in aerospace systems, concerning the reconfiguration of the scheduling of the system applications at execution time, and the update of application software components hosted in partitions, and; (iii) the improvement of the AIR architecture with the safe reconfiguration and update capabilities.

7.1 Future work

Future development concerns the update of a partition while maintaining it in the active mode, i. e., in which partition processes are being executed [4, 52]. This requires the proper implementation of dynamic memory allocation services, which would require deep analysis and more time. The ability to update software components of a partition in the active mode is deemed appropriate to the update of critical spacecraft functions, namely avionics. The update of a partition in the active mode may further need to cope with update specific parts of a partition (e. g., a software procedure), which may benefit from the developments in dynamic software update area [51, 50] (Chapter 2), although applying the referenced methodologies to TSP systems.

The spacecraft computational system may benefit of roll-back update features to allow flexible and robust replacement of the original partition software or configurations, for example to replace the original application after a certain mission phase in order to spare memory resources or reconfigure system parameters according to old configurations. Although, this may require the implementation of a partition version control system thus introducing additional complexity to the spacecraft software. Therefore, it should be studied how to incorporate this feature in spacecraft computational system, and how to implement it in an efficient manner.

Besides the focus on the update of applications in the context of partitions, it may be useful to proceed with updates of the operating system kernel itself, without modifying the partition applications. The update of common operating systems, such as Linux, without the need to restart the execution has been studied and applied to diverse purposes [68, 69, 70, 71]. However, the adaptation of common techniques to the update of operating systems suitable to safety-critical systems, such as those found in the aerospace industry, still requires great efforts in the analysis and design of referred techniques. Other challenges concern the update of operating system kernels in TSP architectures.

Bibliography

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

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

[3] J. Rufino, J. Craveiro, T. Schoofs, C. Tatibana, and J. Windsor, "AIR Technology: a step towards ARINC 653 in space," in *Proc. DASIA 2009 "DAta Systems In Aerospace" Conf.* Istanbul, Turkey: EUROSPACE, May 2009.

[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. Berlin Heidelberg: Springer, 2010.

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

[6] P. Plancke and P. David, "On board computer & data systems," European Space Agency (ESA), Technology Harmonisation - Technical Dossier Issue 1, Revision 2, Feb. 2003.

[7] A. T. Tai, K. S. Tso, L. Alkalai, S. N. Chau, W. H. Sanders, and S. William H., "Low-cost error containment and recovery for onboard guarded software upgrading and beyond," *IEEE Trans. Computers*, vol. 2, pp. 121–137, 2002.

[8] 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.

[9] D. Brown and G. Webster, "Now a Stationary Research Platform, NASA's Mars Rover Spirit Starts a New Chapter in Red Planet Scientific Studies," http://www.nasa.gov/mission_pages/mer/news/mer20100126.html, Jan 2010.

[10] ——, “NASA Trapped Mars Rover Finds Evidence of Subsurface Water,” http://www.nasa.gov/mission_pages/mer/news/mer20101028.html, Oct 2010.

[11] J. Rosa, J. Craveiro, and J. Rufino, “Safe online reconfiguration of time- and space-partitioned systems,” in *9th IEEE International Conference on Industrial Informatics (INDIN'2011)*, Caparica, Lisboa, Portugal, Jul. 2011, accepted for publication.

[12] ——, “Adaptability and survivability in spaceborne time- and space-partitioned systems,” in *EUROCON 2011 - International Conference on Computer as a Tool*, Lisboa, Portugal, Apr. 2011.

[13] ——, “Exploiting AIR composability towards spacecraft onboard software update,” in *Actas do INForum - Simpósio de Informática 2010*, Braga, Portugal, Sep. 2010.

[14] J. Rufino, J. Rosa, and J. Craveiro, “Desenvolvimento e actualização de software para sistemas aeroespaciais em arquitecturas compartimentadas,” Faculdade de Ciências da Universidade de Lisboa, AIR-II Technical Report RT-10-10, Oct. 2010.

[15] J. Rosa, J. Craveiro, and J. Rufino, “Challenges in the design and development of spacecraft onboard software update,” Faculdade de Ciências da Universidade de Lisboa, AIR-II Technical Report RT-10-12, Nov. 2010.

[16] J. R. Wertz and W. J. Larson, Eds., *Space Mission Analysis and Design*, 3rd ed. Microcosm Press and Kluwer Academic Publishers, 1999.

[17] D. Doody, “Basics of Space Flight,” http://www2.jpl.nasa.gov/basics/bd.php, Nov 2010, NASA Jet Propulsion Laboratory, California Institute of Technology.

[18] P. Fortescue, J. Stark, and G. Swinerd, Eds., *Spacecraft Systems Engineering*, 3rd ed. John Wiley and Sons, 2003.

[19] “Fault-detection, fault-isolation, and recovery (FDIR) techniques,” NASA, Johnson Space Center (JSC), Tech. Rep. Technique DFE-7.

[20] P. Rathsman, J. Kugelberg, P. Bodin, G. D. Racca, B. Foing, and L. Stagnaro, “Smart-1: Development and lessons learnt,” *Acta Astronautica*, vol. 57, no. 2-8, pp. 455–468, 2005, infinite Possibilities Global Realities, Selected Proceedings of the 55th International Astronautical Federation Congress, Vancouver, Canada, 4-8 October 2004.

[21] A. Meyer, "Genesis: Search for Origins," http://genesismission.jpl.nasa.gov/gm2/spacecraft/index.html, nov 2009.

[22] O. Camino, M. Alonso, D. Gestal, J. de Bruin, P. Rathsman, J. Kugelberg, P. Bodin, S. Ricken, R. Blake, P. P. Voss, and L. Stagnaro, "Smart-1 operations experience and lessons learnt," *Acta Astronautica*, vol. 61, no. 1-6, pp. 203–222, 2007, bringing Space Closer to People, Selected Proceedings of the 57th IAF Congress, Valencia, Spain, 2-6 October, 2006.

[23] J.-R. C. Cook, "Engineers Diagnosing Voyager 2 Data System," http://www.jpl.nasa.gov/mobile/news/index.cfm?release=2010-151, May 2010, NASA Jet Propulsion Laboratory, California Institute of Technology.

[24] J. Sølvhøj, M. Breiting, and M. B. Thomsen, "Onboard computer for pico satellite," Technical University of Denmark, Tech. Rep., Jan 2002.

[25] CPUShack.Net, "The CPUs of Spacecraft Computers in Space," http://www.cpushack.com/space-craft-cpu.html, 2005.

[26] C. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *Digital Avionics Systems Conference, 2007. DASC '07. IEEE/AIAA 26th*, Oct. 2007, pp. 2.A.1–1 –2.A.1–10.

[27] "Test method standard, microcircuits," Department of Defense, United States of America, Tech. Rep. MIL-STD-883, Feb 2010.

[28] G. D. Racca, A. Marini, L. Stagnaro, J. van Dooren, L. di Napoli, B. H. Foing, R. Lumb, J. Volp, J. Brinkmann, R. Grünagel, D. Estublier, E. Tremolizzo, M. McKay, O. Camino, J. Schoemaekers, M. Hechler, M. Khan, P. Rathsman, G. Andersson, K. Anflo, S. Berge, P. Bodin, A. Edfors, A. Hussain, J. Kugelberg, N. Larsson, B. Ljung, L. Meijer, A. Mörtsell, T. Nordebäck, S. Persson, and F. Sjöberg, "Smart-1 mission description and development status," *Planetary and Space Science*, vol. 50, no. 14-15, pp. 1323–1337, 2002.

[29] J. Andersson, J. Gaisler, and R. Weigand, "Next generation multipurpose microprocessors," in *Proceedings of the DASIA 2010 "DAta Systems In Aerospace" Conference*. Budapest, Hungary: EUROSPACE, Jun 2010.

[30] "Avionics Architectures, ULC - Avionics Overview," Dec 2008, Spacebel, University of Liege.

[31] IEC, "IEC 60027-2: Letter symbols to be used in electrical technology - Part 2: telecommunications and electronics," aug 2005, International Electrotechnical Commission (IEC).

[32] H. Silva, A. Constantino, M. Coutinho, D. Freitas, S. Faustino, M. Mota, P. Colaço, J. Sousa, L. Dias, B. Damjanovic, M. Zulianello, and J. Rufino, "RTEMS CENTRE - Support and Maintenance CENTRE to RTEMS Operating System," in *Proc. DASIA 2009 "DAta Systems In Aerospace" Conf.* Istanbul, Turkey: EUROSPACE, May 2009.

[33] H. Malcom and H. K. Utterback, "Flight software in the space department: A look at the past and a view toward the future," *Johns Hopkins APL Technical Digest*, vol. 20, no. 4, pp. 522–532, 1999.

[34] M. Pignol, "Cots-based applications in space avionics," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10. European Design and Automation Association, 2010, pp. 1213–1219.

[35] "Apollo 14 mission report," NASA, Manned Spacecraft Center, Houston, Texas, Tech. Rep. NASA MSC-04112, May 1971.

[36] N. G. Leveson, "The role of software in spacecraft accidents," *AIAA Journal of Spacecraft and Rockets*, vol. 41, pp. 564–575, 2004.

[37] A. Ellery, J. Kreisel, and B. Sommer, "The case for robotic on-orbit servicing of spacecraft: Spacecraft reliability is a myth," *Acta Astronautica*, vol. 63, no. 5-6, pp. 632–648, 2008.

[38] 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.

[39] A. J. Kornecki and J. Zalewski, "Certification of software for real-time safety-critical systems: state of the art," *ISSE*, vol. 5, no. 2, pp. 149–161, 2009.

[40] G. C. Necula and P. Lee, "Safe kernel extensions without run-time checking," in *Proc. USENIX 2nd Symposium on Operating Systems Design and Implementation*, 1996, pp. 28–31.

[41] M. Neukirchner, S. Stein, H. Schrom, and R. Ernst, "A software update service with self-protection capabilities," in *DATE*, 2010, pp. 903–908.

[42] MIRA Limited, *MISRA-C: 2004 Guidelines for the use of the C language in critical systems*, 2004th ed. Watling Street, Nuneaton, Warwickshire CV100TU, UK: MIRA Limited.

[43] S. Montenegro, S. Jähnichen, and O. Maibaum, "Simulation-based testing of embedded software in space applications," in *Workshop: Embedded Systems*

- Modeling , Technology and Applications 2006, I. 1-4020-4932-3 Springer, Ed., 2006.

[44] 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.

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

[46] L. Sha, "Dependable system upgrade," in *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 1998, p. 440.

[47] K.-F. Ssu and H. Jiau, "Online non-stop software update using replicated execution blocks," in *Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International*, 2000, pp. 319 –324.

[48] H. Seifzadeh, A. Kazem, M. Kargahi, and A. Movaghar, "A method for dynamic software updating in real-time systems," in *Proceedings of the 8th IEEE/ACIS International Conference on Computer and Information Science*, Shanghai, PR China, Jun. 2009.

[49] S. M. Ellis, "Dynamic software reconfiguration for fault-tolerant real-time avionic systems," *Microprocessors and Microsystems*, vol. 21, pp. 29–39, 1997.

[50] M. Wahler, S. Ritcher, and M. Oriol, "Dynamic software updates for real-time systems," in *Proceedings of the HotSWUp'09*, Orlando, Florida, USA, Oct. 2009.

[51] M. Hicks, "Dynamic software updating," *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 6, pp. 1049–1096, Nov. 2005.

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

[53] 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.

[54] AEEC (Airlines Electronic Engineering Committee), "Avionics application software standard interface, part 2 - extended services," Aeronautical Radio, Inc., ARINC Spec. 653P2-1, Dec. 2008.

[55] J. Craveiro and J. Rufino, "Schedulability analysis in partitioned systems for aerospace avionics," in *Proc. 15th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA 2010)*, Bilbao, Spain, Sep. 2010.

[56] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a flexible real time scheduling framework," *Ada Lett.*, vol. XXIV, no. 4, 2004.

[57] D. W. Lewis, *Fundamentals of Embedded Software: Where C and Assembly Meet*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[58] P. Pushner and C. Koza, "Calculating the maximum execution time of real-time programs," *Journal of Real-Time Systems*, vol. 1, pp. 160–176, Sep. 1989.

[59] A. Colin and I. Puaut, "Worst-case execution time analysis of the rtems real-time operating system," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, ser. ECRTS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 191–.

[60] "Hard real-time system development," in *Embedded Systems Design*, ser. LNCS, B. Bouyssounouse and J. Sifakis, Eds., vol. 3436. Berlin Heidelberg: Springer-Verlag, 2005, pp. 10–14.

[61] 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.

[62] 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.

[63] *RTEMS C User's Guide*. On-Line Applications Research Corporation, 2006, 4.8.0.

[64] 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.

[65] 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.

[66] G. Industries, "Galileo software standards (GSWS)," 24/05/2004.

[67] T. Schoofs, S. Santos, C. Tatibana, J. Anjos, J. Rufino, and J. Windsor, "An Integrated Modular Avionics development environment," in *Proceedings of the DASIA 2009 "DAta Systems In Aerospace" Conference*. Istanbul, Turkey: EUROSPACE, May 2009.

[68] K. Makris and K. D. Ryu, "Dynamic and adaptive updates of nonquiescent subsystems in commodity operating system kernels," in *In EuroSys'07 Conf*, Lisboa, Portugal, Mar. 2007.

[69] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *In EuroSys'09*, 2009.

[70] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "Opus: Online patches and updates for security," in *In 14th USENIX Security Symposium*, 2005, pp. 287–302.

[71] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew, "Live updating operating systems using virtualization," in *Proceedings of the 2nd international conference on Virtual execution environments*, ser. VEE '06. New York, NY, USA: ACM, 2006, pp. 35–44.