Formal Specification of the ARINC 653 Architecture Using *Circus*

ARTUR OLIVEIRA GOMES

Submitted for the degree of Master of Science by research

THE UNIVERSITY OF YORK
DEPARTMENT OF COMPUTER SCIENCE

March 2012

To Lidiane and my parents.

Abstract

Nowadays, the avionics industry is moving towards the use of a new architecture for aircraft systems called Integrated Modular Avionics (IMA), which consists of a distributed, highly integrated platform, in which a variety of applications can be executed in the same hardware. Standards and guidelines for the development of aircraft systems now provide guidance for using formal methods during the development process of these systems. In this dissertation we present an approach for using *Circus* as a formal language to model and validate the IMA architecture focusing on temporal partitioning. We provide here an overview of the IMA architecture, detailing its components and features. We also present an overview of existing approaches on certification and verification of IMA systems. We also present a brief survey of formal languages for the design of concurrent systems. Later, we present a *Circus* model of three layers of components of the IMA architecture. It comprises a model of the operating system layer, the application executive (APEX) layer and the partitions layer. Moreover, we validate the *Circus* model by translating it into CSP with time constructs and using the model checker FDR. Finally, we conclude this dissertation with our plans for future work.

Contents

List of Figures

Acknowledgements

I would like to start thanking my parents for all that they have given me and all that they have sacrificed for me to chase my dream. Thanks to my parents for giving me the opportunity to study abroad. For their continuous support and motivation, and for making me feel safe and comfortable, even far away from home. I will be eternally grateful to them.

Though it will not be enough to express my gratitude in words to my beloved fiancé, Lidiane, for her courage, love, and care for me. After almost one year being physically distant but emotionally present, she left her job and parents to come to England, just to be close to me. And she came exactly when I needed her the most. We shared moments that we'll never forget.

I would also like to thank my supervisor, Dr. Ana Cavalcanti for continuous support in developing my research project. Her insights helped me to grow as a researcher and a person. Thanks to my examiner, Professor John Clark, whose suggestions helped me improve my dissertation.

I would like to express my gratitude to my undergraduate research supervisor, Dr. Marcel Oliveira, for introducing me to the formal methods. He helped me to grow as a researcher for many years, and made me look beyond my limitations, always helping to remind me how far I've come, and motivating me to go further and work harder than I even thought I could. He taught me the basis of the technical knowledge needed in order to produce the results presented in this thesis, and for that I'll always be grateful to him.

Many thanks to André Freire, Alvaro Miyazawa, Sara Melo, Miguel Prôa and Pedro Ribeiro, for being good friends. I'll always remember the pleasant moments we shared together. Special thanks to André Freire, for helping me since my arrival in York and for being a very good friend since then. I would also like to thank Alvaro Miyazava, for being patient and generous with his time in helping me with technical issues during my research, and for all the time we spent together. Thanks for Mrs Filomena Ottaway, formerly Research Study Administrator, for all her help and assistance, since the day of my application to the Department of Computer Science, and for being a good friend since then. Finally, many thanks to my workmates, Chris Marriott, Frank Zeyda, Kun, and Alvaro, for the friendly and supportive work environment.

Author's declaration

I hereby declare that the contents of this thesis are the result of my own original contribution, except where otherwise stated.

Chapter 1

Introduction

Nowadays, the avionics industry is moving towards the use of a new architecture for aircraft systems called Integrated Modular Avionics (IMA). This new architecture consists of a distributed, highly integrated platform, in which a variety of applications can be executed in the same hardware, promoting reductions of power requirements, weight and costs of maintenance. Standards and guidelines for the development of aircraft systems now provide guidance for using formal methods during the development process of these systems.

In this dissertation, we present an approach for using formal methods, specifically, *Circus* as a formal language to model the three top levels of the IMA architecture: the operating system, the ARINC API module, known as the Application Executive (APEX), and the partitions. We present here in this chapter the motivation and objectives of our work.

1.1 Motivation

In the past decades, aircraft systems have been produced using a federated architecture. By producing equipment using that architecture, airframers need to consider the production of their own computer processor architecture, including hardware, connectors, and operating system for that architecture, as well as the software for the specific avionic applications. The search for reduction of costs of production, weight and power consumption has been a challenge.

In order to address those needs, the Integrated Modular Avionics (IMA) architecture was proposed. The IMA architecture consists of a distributed system, where many aircraft applications can be executed in the same hardware module, sharing computing resources, communications and input and output devices.

The guidelines for the IMA architecture have been developed by Aeronautical Radio, Incorporated (ARINC) [5]. Considerations for implementation of hardware and software are split into dozens of sections, for example the ARINC 653 [2] document, presents the components of the interface among IMA applications and the operating system. According to the DO-178B document, one of the objectives of the verification process is to ensure that the resulting software is in conformity with its requirement standards; for IMA, this means the use of the ARINC standards.

An operating system is used to manage the access by the applications to the computing resources, such as, processing unit, memory and sensors. The operating system of the IMA architecture is designed in such a way to prevent, through the concept of partitioning, direct communication among applications. It ensures that none of the partitions can share the same memory area or processing time slice. A module, in accordance with the ARINC standards, may have different architectural configurations with respect to the number of applications running, scheduling for these applications and recovery actions in case of failures.

Cook and Hunt discuss in [12] software reuse in Integrated Modular Avionics. They suggest that if software can be re-used, it can reduce costs since existing software which meets their certification requirements does not need to be verified again. This is relevant if additional features are included in existing software, or if the certification requirements have not changed and hardware upgrade is needed, or the application is to be implemented in a different aircraft model. The ARINC 653 standard addresses software reuse to reduce the verification and validation effort using the concept of partitioning. This work is a motivation for our approach, as we propose here a specification and validation of fixed components of the IMA architecture: the components will be validated only once.

There are many works on certification aiming at reliability and safety of IMA applications. For example, in [28], Hollow et al. discuss certification issues regarding multiple possible architectural configurations of IMA modules. They present a strategy that consists in identifying key components of different IMA configurations and establishing equivalence between them. Our work is independent of the architectural configuration of the module: following the ARINC standards, our models specify that the execution of any two partitions can not overlap each other, according to its predefined scheduling.

The increasing use of computing software in modern aircraft for the past thirty years has motivated regulatory authorities to produce standards and guidelines to be used by aircraft system developers in order to achieve airworthiness certificates. The Radio Technical Commission for Aeronautics (RTCA) published the Software Considerations in Airborne Systems and Equipment Certification (DO-178) [1], providing guidelines for the development and verification of software of different criticality levels.

According to the DO-178B, software verification is one of the main steps during the software development process. The software verification process aims at ensuring that defined system requirements, such as functional and safety requirements, are met in the produced source code. At the moment of its release, in 1992, the DO-178B defined the concepts of verification as reviews, analysis and tests. As the use of formal methods and existing tools was relatively small, it suggested that they can be used in combination with tests.

In the past ten years, formal methods and its tools have already reached a level of maturity to make it possible for them to be applied in industrial scale critical software. For example, Airbus researchers have been using formal methods to analyse their programs [61] where tools such as Caveat are used to perform fully automatic data and control flow analysis and prove user-specified properties from C programs.

The use of formal methods in industry is a new trend, and as a consequence of its acceptance, RTCA has recently released a draft of the forthcoming DO-178C document, providing guidance for using formal methods to be applied during the software life cycle. The use of formal methods, according to DO-178C, can include the development of formal models, using some formal notation, and reasoning about these models, referred in the document as formal analysis. Moreover, the development and use of tools for formal verification is recommended by DO-178C.

In order to formally verify systems, we first need to produce a formal specification of it, in which we can describe the system and its desired properties, for example, the behavioural and timing properties of the system. A formal specification is designed using specification languages, where systems are described in a higher level than any programming language. Specification languages syntax and semantics are defined with a strong mathematical background.

Data aspects of the system can be modeled using state-based languages, such as Z [64], B [3] and VDM [20]. However, state-based languages can not be conveniently used to model behavioural aspects of the system, such as communication between components. The consistency of specifications using the mentioned languages can be validated through theorem proving and model checking. For example, specifications in Z can be validated using the theorem prover ProofPower-Z [32] and Symbolic Analysis Laboratory (SAL) [60]. Moreover, we are also able to verify specifications in the mentioned languages through refinement.

Behavioural aspects of a system can be described by using languages such as CSP (Communicating Sequential Processes) [27, 49] and CCS (Calculus of Communicating System) [37]. These languages allow us to describe interactions, communications and synchronisation between processes. But by using languages such as CSP and CCS, the description of complex data aspects of the system becomes inconvenient.

As we intend to specify complex systems, in which it is necessary to describe both data and behavioural aspects, we need to look for a formalism that combines both aspects and allow us to reason about such systems. We can overcome this problem with the help of combinations of the above presented languages. For instance, B and CSP are integrated in [63, 6]. Z combined with CCS is presented in [21, 62]. Its combination with CSP is considered in [39, 50]. Woodcock and Cavalcanti define *Circus* [65], which is a formalism that combines not only Z, CSP, but also Morgan's refinement calculus [38] and Dijkstra's language of guarded commands [16]. Its semantics is defined using the Unifying Theories of Programming [26]. In addition, the description of timing properties of the specification can be modelled by using *Circus Time* [55]. Moreover a refinement calculus for *Circus* is presented in [42] with tool support [66] using ProofPower-Z [32].

By using *Circus*, we can profit from its support for concurrent systems, which is one of the characteristics of IMA programs. We can benefit from the use of the *Circus* refinement calculus to model a system at different abstraction levels, and, by using its refinement laws, verify the consistency of the different refinement levels with the help of formal proofs.

In the context of Integrated Modular Avionics, there are some works in which formal methods are used for the verification of IMA components. For example, Delange et al. [14], focus on the verification of elements of the ARINC 653 architecture, such as the set of operating system services available for the applications through the APEX, which is an application programming interface (API) that offers services from the operating system to the partitions and is defined as a layer between the operating system and the partitions. After formal verification of the aspects of the designed architecture, the approach allows code generation, including the implementation of the APEX services. The generated code of the components of the architecture is compiled together with externally generated IMA application source code. The binaries are simulated in POK, an in-house produced runtime system.

A different approach is suggested in [22], where a modelling framework for IMA applications called MIMAD, using the synchronous language SIGNAL and the POLYCHRONY toolset, is proposed. The framework consists of a set of tools, such as, an interface for software design, a code generator from the SIGNAL specification, compiler, formal verification tools and model checker. The interface for software design allows the user to model ARINC 653 processes contained within a partition. The overall goal of the toolset is to provide a high-level abstraction in system design; designers use the MIMAD notation.

An approach for the verification of spatial partitioning for IMA applications is presented by Di Vito [15] where noninterference models are adopted. The main goal of this work is to analyse how applications originally designed for a federated architecture behave when migrated into the integrated architecture. The intention is to rule out any observable behaviour of the system in the integrated architecture that can not be reproduced in the federated architecture. The verification of the models is performed using *Prototype Verification System* (PVS) with theorem proving support for a few different scenarios as examples. The approach is very relevant; however, it covers only memory aspects of partitioning for the IMA architecture.

In this dissertation we present a formal specification and validation of the three top layers of the IMA architecture using *Circus*. It comprises the operating system, the Application Executive (APEX) and the partitions allocated to execute within the IMA module.

This work differs from the literature since we aim at formal modelling of the components of the architecture, focusing on capturing temporal partitioning of IMA: timing properties of the architecture for scheduling the execution of the partitions according to the ARINC 653 standards document.

The internal execution of the tasks within a partition, ARINC processes, is not captured in our model. We capture the behaviour of the system in which partitions can request the use of the APEX services regardless from which ARINC process the request was originated. Moreover, our model captures the data aspects of the configuration tables, as specified in the ARINC 653 document, in which the number of partitions, predefined partition scheduling and recovery actions are specified.

1.2 Objectives

Our main goal in this dissertation is to formally specify and validate the three top levels of the IMA module; we capture the basic services offered by the operating system of an IMA module, as specified in the ARINC 653 document. Our model covers the interactions between the operating system and IMA applications, which are allocated into partitions within a hardware module. Scheduling capabilities of the ARINC 653 are modelled using *Circus Time* constructs.

Some requirements of the IMA programs, such as memory space and processing time, are listed in the configuration tables. These tables are produced by the system integrator. In our work, we do not focus on memory requirements of the applications, but only time.

To validate our *Circus* model of the IMA architecture we use FDR [30]. We produce a CSP model of our *Circus Time* model, and then, check some properties of our model such as deadlock-freedom and livelock-freedom. As the CSP language itself does not support timed processes, we use approaches that allow us to include time constructs from our *Circus Time* model into CSP.

The contribution of this work is to have a formal model of the IMA architecture in which the interactions between the IMA application and the operating system, managed through the services of the Application Executive (APEX), are available for a single IMA application at the same time. Using *Circus Time* constructs, we are able to specify the behaviour of the operating system, which uses time to determine whether or not each of the partitions of the module are going to execute, according to the scheduling table defined in the configuration tables. We also can prevent two partitions overlapping each other, with respect to the temporal partitioning, with *Circus Time* constructs such as timeouts and interrupts.

The work presented here is relevant since we make use of formal methods in order to describe a critical system that involves time management during its execution. We aim at providing a formal model, in conformity with the ARINC 653 standards document, in which its behaviour is modelled in such a way that allows us to capture the temporal partitioning properties of the system. The model to be presented in the following chapters supports different configuration of IMA modules since we describe the architecture in *Circus*, including the data structure of the configuration tables.

1.3 Structure of the dissertation

In Chapter 2 we present the Integrated Modular Avionics architecture, describing the components of the architecture. Moreover, we also present the characteristics and restrictions of the ARINC 653 specification. We also review related approaches of verification of IMA and discuss similarities and open problems. We conclude Chapter 2 presenting an overview of *Circus* providing an example of a specification in that language.

In Chapter 3 we present how we formalise the IMA architecture and available services in *Circus Time*, based on the ARINC 653 specification. We define a *Circus Time* process for each level of the architecture: the operating system, the APEX and the partitions. We also present how we model the interaction between each components of the architecture and how the execution of the partitions is managed by the operating system during the lifetime of the module.

The translation from *Circus Time* to CSP is presented in the Chapter 4, along with the decisions made in order to represent the equivalent model in CSP which includes the state variables of the *Circus Time* processes.

Chapter 5 closes this dissertation with conclusions and lessons learned. Moreover, we briefly discuss possible directions to future contributions from the current stage of this work. For instance, we present some steps towards the model generation strategy from IMA programs to *Circus*.

Chapter 2

Background and Related Work

In the past few years, management software for commercial aircraft has been developed to be executed under dedicated hardware, usually based on the use of a federated architecture. The use of such architecture avoids error propagation: each component of the airframe, such as aircraft engine controller and autopilot, is executed in its own module and the interaction with other modules is very restricted. On the other hand, some of the consequences of using federated architecture are the high amount of redundant hardware, cables, and high power consumption. Moreover, it is likely that the replacement or upgrade of obsolete software may become difficult and expensive over the years. This became a motivation for aircraft producers to move towards an architecture where several applications are able to run under the same hardware module, sharing its computing resources. This new architecture is known as *Integrated Modular Avionics* (IMA).

In this chapter we present the components of the IMA architecture along with a survey of approaches for certification of avionics systems as well as formal languages and verification approaches.

2.1 Integrated Modular Avionics

The IMA architecture is motivated by the possibility of providing a higher integration between programs used for controlling aircraft equipments. The idea is to have a distributed, flexible and reusable architecture, where applications of different criticalities can be executed concurrently in the same module.

Instead of having multiple processing units, where each one is allocated in one module and executes a specific task, the IMA architecture uses an operating system to manage the execution of applications running in a same IMA module. In the commercial aircraft industry, the IMA architecture has been designed in accordance to the ARINC standards.

The advantages of the adoption of the IMA architecture are the reduction of hardware components and cabling, and weight, as well as power consumption. It is possible since the IMA programs run in the same equipment, sharing resources, and reducing the time used for communications between the applications. Communications between applications running within the same module are managed by the operating system, reducing considerably the time and wires used. Moreover, communications between modules, displays, sensors and actuators are carried out within the components by a common bus.

The architecture is organised in levels, from hardware to software, with an operating system, as illustrated in Fig. 2.1. That feature allows the hardware designer to produce their equipment without taking into account how the designed software behaves in its equipment. On the other hand, the IMA architecture allows software developers to freely produce their applications, according to the ARINC 653 standards, regardless of the hardware equipment it will run into. Moreover, the structure of the IMA architecture allows aircraft producers to use commercial off-the-shelf components, avoiding obsolescence. Thus, any replacement or upgrade of both hardware and software can happen independently from each other, reducing costs of maintenance.

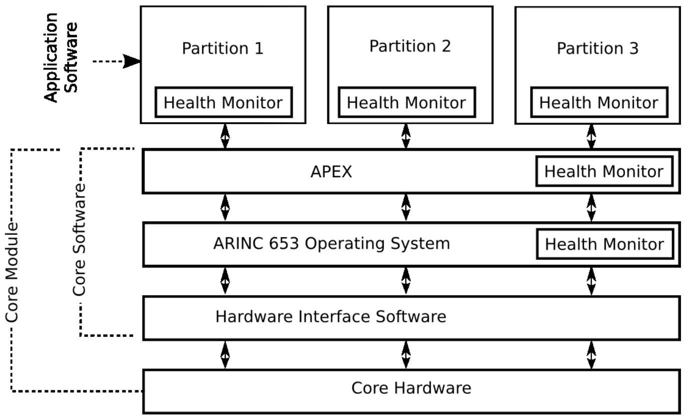

Figure 2.1: ARINC 653 module architecture [2]

The key feature to avoid fault propagation is the concept of partitioning: each application executed in an IMA module is allocated into a *partition*. In IMA, partitions are isolated from each other by using the concept of temporal and spatial partitioning. Temporal partitioning means that each partition has a slice of time dedicated to execute its computations and communications, according to the requirements of the application allocated in the partition. Likewise, spatial partitioning provides a dedicated portion of the module memory for each partition.

In order to allow the IMA applications to run simultaneously in the same module, basic services are provided by an application programming interface between the operating system and the partitions, known as *Application Executive* (APEX). These services allow each partition to manage its own tasks, ARINC 653 processes, as well as communicate with the other partitions through requests to the operating system via the APEX services.

Errors and failures detected during the IMA module execution are managed by the health monitor, which is able to provide the correct recovery action for such problems. The health monitor can distinguish the recovery action depending on the kind of effect that the error can cause within the module, acting either within the executing partition boundaries or within the entire module.

In order to establish the time and memory boundaries of each partition to be executed within an ARINC 653 module, a configuration table for each particular set of partitions is defined by the system integrator. A configuration table is defined for each module and depends on the number of partitions to be executed and their particular requirements, such as time and memory space. Moreover, the configuration tables also contain the schedule of the partitions execution, based on the requirements of each partition.

In the next subsections we describe in detail the concepts of partitions, scheduling, health monitor and the configuration tables.

2.1.1 Partitions

Each partition has a set of fixed attributes such as name and identifier. A portion of the total memory space of the module is dedicated to each partition. Each partition has access to the resources available in the module for a predefined amount of time, according to the schedule in the configuration tables. Moreover, the ARINC 653 defines services for communication within a partition, between the partitions within the module, and also between partitions allocated in different modules.

Processes

A partition contains one or more ARINC 653 processes, which may operate concurrently in order to execute the functionalities of its partition. Processes are created and initialised during the partition initialisation. In a partition restart, the processes are recreated, since the partition starts again with the initialisation phase. Partitions are able to restart one or more of its processes. Moreover, a partition may be able to respond to faults and failures of a process, by restarting or terminating such processes according to the rules defined by the health monitor in the partition level.

In order to solve an ambiguity problem, we use the convention that when we mention a process in this dissertation, we are referring to an ARINC 653 process. However, as we are specifying the ARINC 653 architecture using *Circus*, which also uses the term process, we hereafter refer to these as *Circus* processes.

Each process of a partition has its own name, entry point (which is a memory address of the memory portion allocated to the partition), as well as a portion of the total time dedicated to the partition to be executed. Moreover, each process has a period of execution, deadline, and priority of execution. As the partition has a scheduling policy to execute the processes, each process can be in a number of different states: dormant, ready, running and waiting. A process inside a partition is not visible outside the partition. The behaviour of the process is managed by its partition. The management of a process inside a partition is made using the APEX services.

Some of the attributes of a process, such as name, period and entry point are statically defined, and therefore cannot be changed after the partition initialisation. Differently from the fixed attributes, variable attributes, such as process state, may change during the execution of the processes in the partition via the APEX services requests.

Partition Communication

The communication between partitions, referred in the ARINC 653 specification as inter-partition communication, is made using messages. Messages are sent from one source to one or more destinations. Message transmission is atomic: a partial message is not delivered to the destination. As illustrated in Fig. 2.2, interpartition communication allows the exchange of messages between partitions within the same ARINC 653 module, between different ARINC 653 modules, and with other non-ARINC 653 modules.

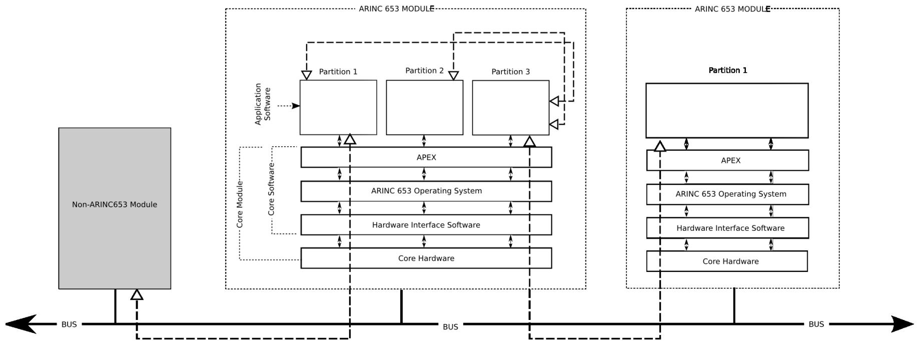

Figure 2.2: ARINC 653 - Interpartition communication example

Each partition has its own port, responsible for sending and receiving messages. A port using sampling mode is usually used to send and receive the same message with updated data. In this mode, a message to be sent remains in the port until it is overwritten by a new occurrence of the message. Similarly, a received message remains in the port until a new message is received, being replaced by the newer message. Differently from the sampling mode, the queueing mode uses a queueing policy for transmitting messages. In this mode, no message is replaced, preventing loss of data. Messages are buffered in a queue and are transmitted according to a FIFO order.

The operating system has records of each port associated with the partitions. Moreover, all communication between partitions is made through requests to the APEX services.

2.1.2 Schedules

Each partition in the module has access to one of the processors within a time window. A time window is a portion of the major time frame, the length of the cycle, which is the time required to process requests from all partitions in the module. As the execution of all partitions is cyclic, after executing all processes within the major time frame, the scheduler of the operating system starts a new cycle with the execution of the partition allocated to its first time window.

For each partition, the time window offset is the delay between the beginnings of the major time frame until the execution of the partition. Moreover, if the execution of a partition is periodic, the period is the time between the initial moment of the execution of the first occurrence until the initial moment of the next execution of the partition. Finally, a spare time occurs when no partitions are executing, for example, if the next partition to be executed is periodic and there is still a delay until its execution. Fig. 2.3 gives an example of the order of execution of three partitions, *Partitions A*, *Partition B* and *Partition C*, where both *Partition B* and *Partition C* are executed twice within the major time frame. After the second execution of *Partition C*, the scheduler starts again the execution of *Partition A*.

The schedule of partition execution is defined in the configuration tables, in which the system integrator defines the required amount of time for each partition and the order of execution based on the requirements of each application allocated into a partition.

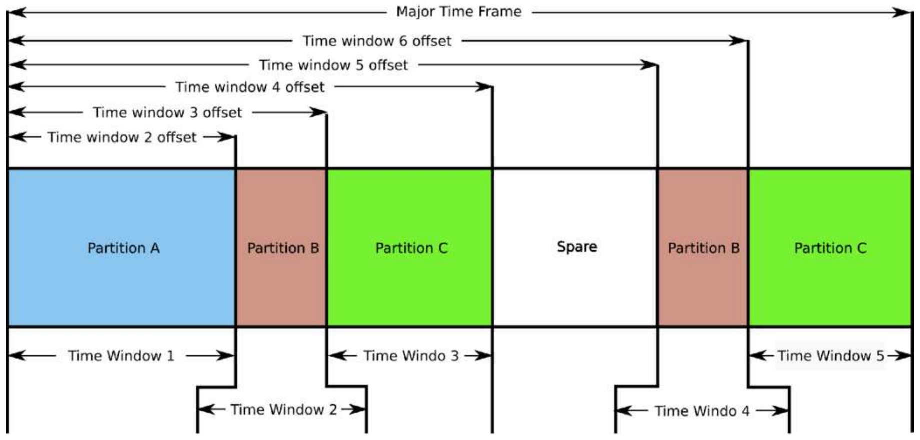

Figure 2.3: ARINC 653 - Partition time window example [2]

2.1.3 Health Monitor

The health monitor is an essential mechanism used in the ARINC 653 module. It is responsible for handling errors and failures raised during the execution of the module. A set of recovery actions is defined in the configuration tables. There are three different levels of health monitor within a module: module, multi-partition, and partition.

In the module level, the health monitor manages the errors regarding the execution of the module itself, when the error raised is not related to the partition currently being executed. The health monitor can decide whether or not the module should be reinitialised, switched off, or if the error must be ignored.

In the multi-partition level, the set of recovery actions are used to decide whether or not the partition in execution can compromise the module. In this level, the health monitor can decide if the decision taken affects only the partition or the module.

Finally, the partition level health monitor responds to errors specific to the execution of processes within a partition. Raised errors can lead the health monitor to reinitialise the partition or set it to idle.

2.1.4 Configuration Tables

The configuration tables are an essential feature of the ARINC 653 module: they are used for describing the structure of the module. The ARINC 653 standard specifies that configuration tables are described using Extensible Markup Language (XML) schemas. These tables are produced by the system integrator, which is responsible for collecting the requirements of each partition allocated within the module and to provide means for their execution, such as a schedule of partition execution with the required amount of time for each partition. Any particular set of applications may be executed in any ARINC 653 module, once the requirements for its execution are predefined in the configuration tables.

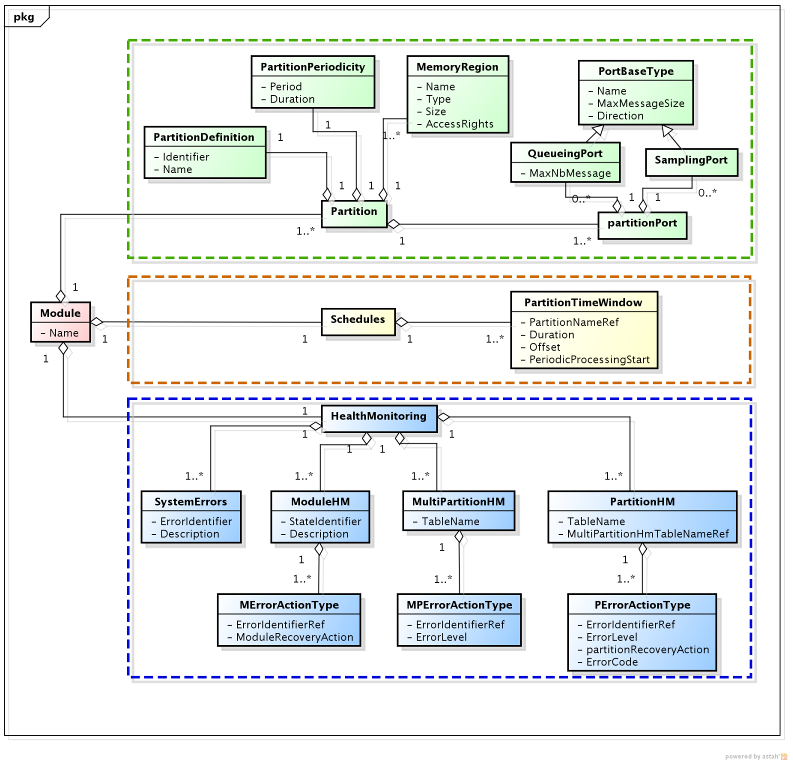

Figure 2.4: ARINC 653 configuration tables structure [2]

The configuration tables are split in three blocks, as illustrated in Fig. 2.4. As an ARINC 653 module can have more than one allocated partition, the section *Partition* (in green) provides information about the requirements of each partition within the module, such as time and memory requirements. Moreover, the requirements for communication between partitions, such as port type, message size, and the name of the ports for each partition are also provided in the *Partition* section of the configuration tables. The section *Schedules* (in yellow) provides the correct order of execution of the partitions within the module. Finally, the errors and recovery actions performed by the health monitor are also provided in the section *HealthMonitor* (in blue).

An example of an XML configuration table is illustrated below. In this example, a single partition *systemManagement* is allocated within the module, with two memory regions and a single communication port. The schedule of the module is defined by a single partition, the *systemManagement* partition with a time window of 2×10^7 nanoseconds (0.02 seconds). Moreover, the health monitor section of the configuration table example contains the set of recovery actions of the three levels presented in Section 2.1.4, in case a system error is raised during the execution of the module.

```xml
<ar:MODULE Name="ARINC 653 Module" xsi:schemaLocation="ARINC653 ModuleExample.xsd"
        xmlns:ar="ARINC653" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <ar:Partitions>
   <ar:Partition>
     <ar:PartitionDefinition Name="systemManagement" Identifier="1"/>
     <ar:PartitionPeriodicity Duration="20000000" Period="20000000"/>
     <ar:MemoryRegions>
       <ar:MemoryRegion Type="RAM" Size="1048576" Name="mainMemory" AccessRights="READ_WRITE"/>
       <ar:MemoryRegion Type="Flash" Size="524288" Name="Flash" AccessRights="READ_ONLY"/>
     </ar:MemoryRegions>
     <ar:PartitionPorts>
       <ar:PartitionPort>
         <ar:QueuingPort MaxMessageSize="30" Name="Stat_2Dq" MaxNbMessage="30" Direction="DESTINATION"/>
       </ar:PartitionPort>
     </ar:PartitionPorts>
 </ar:Partition>

 <ar:Schedules>
   <ar:PartitionTimeWindow PeriodicProcessingStart="false" Duration="20000000"
                       PartitionNameRef="systemManagement" Offset="0"/>
 </ar:Schedules>

 <ar:HealthMonitoring>
   <ar:SystemErrors>
     <ar:SystemError ErrorIdentifier="5" Description="segmentation error"/>
   </ar:SystemErrors>
   <ar:ModuleHM StateIdentifier="2" Description="system function execution">
     <ar:ErrorAction ErrorIdentifierRef="5" ModuleRecoveryAction="SHUTDOWN"/>
   </ar:ModuleHM>
   <ar:MultiPartitionHM TableName="System partitions HM table">
     <ar:ErrorAction ErrorIdentifierRef="5" ErrorLevel="PARTITION" />
   </ar:MultiPartitionHM>
   <ar:PartitionHM MultiPartitionHMTableNameRef="System partitions HM table" TableName="systemManagement HM table">
     <ar:ErrorAction ErrorIdentifierRef="5" PartitionRecoveryAction="IDLE"
                 ErrorLevel="PROCESS" ErrorCode="MEMORY_VIOLATION" />
   </ar:PartitionHM>
 </ar:HealthMonitoring>
</ar:MODULE>
```

Listing 2.1: Example of an ARINC 653 XML configuration table

2.2 Certification and Verification of IMA systems

In this section we first give an overview of works on certification aiming at reliability and safety of IMA applications. Then, we focus on some works related to the application of formal methods in aircraft systems, especially, those works related to formal verification of IMA systems.

2.2.1 Certification Strategies

A first example of approaches to certification is presented in [11], where Comny et al. present an approach for the development of safety assurance contracts. The analysis starts with the identification of safety dependencies between components and the analysis of the impact to the system, caused by failures in one of these components. Then, the development for a contract for each derived safety requirement is made. A contract is based on a set of rules (preconditions, postconditions, and rely and guarantee conditions) between a server and its clients. The contract development process consist in the analysis of how a component meets its derived safety requirements, which can be performed at four levels, ranging from high-level requirements to a more detailed level such as syntactic, performance or reliability requirements. Among those levels, architectural and behavioural requirements can also be analysed.

Another approach for the development of safety cases for certification of avionics systems is presented in [41]. The goal is to define a method capable of identifying changes in modular applications, in order to facilitate the process of recertification. A way of constructing a safety case for certification is presented, where some aspects of the modification of the application should be analysed. Among these aspects, it is important to analyse if the modified (or new) element conforms with the design rules and API usage criteria for ARINC 653 applications; identify the impact of the modification (or inclusion) of the element in the system; and provide arguments for the certification authorities, such as potential safety implications, and evidence of testing to ensure that the modification does not affect any other application, and any incorrect operation inside the modified application is detected during the initialisation stage. This work adopts the goal structuring notation (GSN) as a way to provide arguments for the certification. As an example in this paper, the GSN structure is used to achieve evidence to the certification of timing characteristics of the integrated modular system.

Another approach for modular certification is presented in [18], where Fenn et al. argues about modular and incremental certification to be applied in aerospace software development. The use of modular and incremental certification is justified by the costs of recertification of changes made. Some steps in order to achieve modular and incremental certification are presented, such as identifying change scenarios (either functional or operational) and identifying dependency-guarantee relationships and dependency-guarantee contracts in order to identify the relations between software elements. Determination of whether a system should be certified using the presented approaches requires aspects such as modularity, degreee of software reuse and level of system complexity to be considered. These criteria should be used to identify whether or not a system should be certified using the presented methods. Moreover, modular certification should be applied during the entire lifetime of legacy systems, specifically when changes are made to these kinds of applications.

Modular certification is also discussed in [53]. Rushby presents an approach which consists in certifying modularised systems not considering the set of all functions as a whole, but to certify the system considering the properties of each functionality in isolation. However, as the system presents interaction between modules and some of those modules depend on other ones, this certification approach should include assumptions of the related modules during the certification of a given module, by using assume-guarantee reasoning. Assume-guarantee reasoning is used for certification instead of its original context, verification.

A strategy of certification of reconfigurable IMA systems is presented in [28]. It is justified by the need to reduce costs of certification. The strategy aims to identify a set of different IMA configurations which can be equivalent, requiring certification of only a single key member of that set. This work differs from ours as we define and validate a unique model of the architecture, independently from the configuration tables, instead of possible architectural configurations.

All the above presented works rely on certification approaches focusing on the analysis of safety requirements. However, software certification does not prove that the system is correct. It only guarantees that the software is in conformance with the standards set by the certification agency. Moreover, we can not analyse whether or not the behaviour of the system meets its specification requirements with certification methodologies. It is, however, possible using formal methods, in which we can analyse the behaviour of the whole system and also the interactions between its components.

In the next subsection we present some verification approaches of avionics systems.

2.2.2 Verification of Avionics Systems

In [22], the Modelling Paradigm for Integrated Modular Avionics Design (MIMAD), a framework for developing IMA applications is presented. It uses the synchronous language SIGNAL [23] and the POLYCHRONY [25] toolset with features like code generation, compiler, formal verification and model checking. However the approach lacks validation of safety properties. The framework includes a graphical user interface which allows the user to easily design IMA applications. Moreover, the concepts of modularity and reusability are present in this work, allowing the reuse of generated models in other contexts. The overall goal of the toolset is to provide a high-level abstraction in system design.

However, the work presented in [22] focuses on modelling only the components of the partition level of the architecture, managing how partitions can access to the APEX services. Differently, in our work, we model the three top levels of the IMA architecture, as illustrated in Fig. 2.1, and focus on temporal partitioning. With temporal partitioning, we model how the operating system manages the execution of partitions with respect to the time slice dedicated to each partition. As a consequence, we can capture how the operating system allows each of the partitions to have access to the APEX services and perform its internal calculations.

The Architecture Analysis and Design language (AADL) is used by Delange et al. in [13] to design ARINC architectures including hardware description. Their models are verified using Real Time Scheduling theory instead of formal methods, focusing on scheduling simulations with support of the Cheddar toolset [29]. Code generation of AADL models is also provided in [13]. The code generated from the AADL model is split in two layers: module and partition. The former provides services related to partition management such as scheduling and interpartition communication, while the latter provides services to the ARINC 653 process management, such as interpartition communication and memory allocation. The generated code from the AADL model of the architecture is compiled together with provided IMA-application source code resulting in binaries files to be executed using the AADL runtime system, POK. The executed binaries are analysed in terms of time isolation and memory constraints. In our approach, the model of the IMA architecture is validated only once.

Moreover, in [14], Delange et al. validates the ARINC 653 architecture using theorems written in Requirements Enforcement Analysis Language (REAL) [52]. With these theorems, they can verify ARINC constraints such as time and memory isolation and also fault coverage.

However, none of the above presented works has presented a formal specification of the IMA architecture that captures the configuration tables data for partitioning management, partitioning scheduling and handling errors.

An approach for the verification of spatial partitioning for IMA applications is presented by Di Vito [15], where noninterference models are adopted. The main goal of this work is to analyse how applications originally designed for federated architecture behaves when migrated into the integrated architecture. The intention is to rule out any observable behaviour of the system in the integrated architecture that can not be reproduced in the federated architecture. The verification of the models are performed using *Prototype Verification System* (PVS) [43] with theorem proving support for a few different scenarios as examples. The approach is related to ours; however, it covers only memory aspects of partitioning for the IMA architecture. However, in our work we focus on temporal aspects of the concept of partitioning for IMA.

Comparing the above presented approaches with the work presented in this dissertation, we do not capture the internal behaviour of the partitions. What we present here is a formal model of the IMA architecture in which partitions are executed according to a schedule and, for each partition in its execution period; a number of services are made available from the Application Executive (APEX). Modelling ARINC 653 processes, process scheduling, intra-partition communication as well as the APEX services for these features, is to be part of our future work.

2.3 Formal Specification Languages

In this dissertation we present a formal specification of the IMA architecture, according to the ARINC 653 standards. We present in this section an overview of the existing formal languages and possibilities of reasoning techniques.

State-based languages, such as Z [64], B [3] and VDM [20], are used to model data aspects of the system. By using these languages, we can provide a mathematical description of systems, using, for example, set theory, first-order logic and lambda calculus. On the other hand, to model behavioural aspects of the system, such as communication between components, using state-based languages becomes inconvenient.

The consistency of state-based specifications can be validated through theorem proving and model checking. State-based verification through refinement is possible for the mentioned languages. A refinement calculus for Z, based on Morgan's work [38], is presented in [9]. Reasoning about the Z specification can be made through theorem proving [32, 54]. The B-method allows the system development through refinement [45], with tool support provided by the B-toolkit [47]. Moreover, proofs obligations regarding refinement can be discharged using a theorem prover, such as Atelier B [10]. Refinement of VDM specifications are presented in [40, 34]. For instance, in [34], VDM specifications are manually translated to PVS [44].

With the help of languages such as CSP (Communicating Sequential Processes) [27, 49] and CCS (Calculus of Communicating System) [37], we can describe interactions, communications, and synchronisation between processes. Notions of refinement are presented in [49] and are supported by a model-checking tool FDR [30]. Moreover, animators such as ProBE [31] are also available for CSP specifications. However, differently from state-based languages, the description of data aspects of the system with the languages mentioned above becomes inconvenient.

As we aim at verifying complex systems, it is unlikely that we can capture both data and behavioural aspects of the system with the above presented formalisms in isolation. We thus need a formalism that can be used to write models that combine both aspects and allows us to prove the refinement of such systems.

Many formalisms have been combined in order to overcome this problem. For instance, combinations of Object-Z [17], an extension of Z that includes notions of object-orientation, with CSP are presented in [58, 19, 33]. A refinement method for [58] is presented in [59]. Mahony et al. merges Object-Z with timed CSP in [35]. B and CSP are integrated in [63, 6]. Z combined with CSP is presented in [39, 50]. Moreover, its combination with CCS is presented in [21, 62].

Woodcock and Cavalcanti define *Circus* [65], which is a formalism that combines not only Z, CSP, but also Morgan's refinement calculus [38] and Dijkstra's language of guarded commands [16]. Its semantics is defined based on the Unifying Theories of Programming [26]. Moreover a refinement calculus for *Circus* is presented in [42] with tool support [66] using ProofPower-Z [32]. An extension of *Circus* for specifying timing aspects of systems, *Circus Time*, is presented in [57, 56].

By using *Circus* we are able to model the IMA architecture, which allows us to model the data aspects of the architecture with using the notion of state, and also we can specify the complex behaviour of the architecture, including *Circus Time* constructs for scheduling of ARINC partitions. Moreover, we are also able to capture concurrency between the ARINC partitions, modelled as *Circus* processes, in parallel between each other. It is also important to mention that it is possible to use the *Circus* refinement calculus [42] in order to prove the refinement between the abstract and more concrete *Circus* models of systems.

2.4 *Circus*

We present in this section a brief overview of the components of the *Circus* notation. We also provide an example of a small specification in *Circus* along with the description of each component.

Circus allows us to specify concurrent systems including data and behavioural aspects. As *Circus* is a combination of Z and CSP, a *Circus* model consists of a sequence of Z paragraphs, such as schemas and axiomatic definitions, channels, channel sets declarations and process definitions. A *Circus* process is composed a state paragraph, a list of actions, and is concluded by the main action of the process. An action can be a schema expression, a command, a call to another previously defined action or a combination of actions using CSP operators such as choice and parallel composition.

We detail the structure of a *Circus* process with the example of a specification of Ping, a computing network service, which calculates the amount of time necessary to send a message to a server and receive a confirmation. The result of the service is a sequence of messages stating whether or not the server has responded to the request, and in case of response, the service can tell the length between the request and the answer. We first define two types used in the specification of the *Ping* *Circus* process. The first one IP_ADDR is an abstract type denoting the IP address of the requested server. Then, we define the type $RESPONSE$, which can be either $RESP$ for a response from the server or a $TIMEOUT$ if the server does not answer within 1000 milliseconds.

$$[IP\_ADDR]$$
$$RESPONSE ::= RESP \mid TIMEOUT$$

We also define a few communication channels for the *Circus* process *Ping*: $send\_data$ and $receive\_data$ are used to communicate with the server, sending and receiving the data for the request; the channel *tick* denotes the passage of time; and $display\_resp$ outputs a sequence containing the result of the request, stating whether each ping request had a reply or a timeout.

$$\mathbf{channel}\ send\_data, receive\_data : IP\_ADDR$$
$$\mathbf{channel}\ tick$$
$$\mathbf{channel}\ display\_resp : RESPONSE \times \mathbb{N}$$

The structure of a *Circus* process is defined as follows. It has a process name and a sequence of paragraphs delimited by **begin** and **end**. Depending on the *Circus* process, it can have parameters, such as the *ip* variable of the *Circus* process *Ping* illustrated below.

$$\mathbf{process}\ \ Ping \ \widehat{=}\ ip : IP\_ADDR \bullet \mathbf{begin}$$

The process may contain a **state**, and a sequence of *Circus* actions, and is concluded with the main action of the *Circus* process, preceded by a '•'symbol. In this example, we present the state *PingSt*, which contains two components: the maximal number of ping requests is stored in max_rep; and the number of requests already performed within the execution of the process is stored in *counter*.

$$\mathbf{state}\ \ PingSt == [max\_rep : \mathbb{N};\ counter : \mathbb{N}]$$

We initialise the state with the *Circus* action *InitSt*, in which the initial value for max_rep is 4, and the *counter* is initially set to 0.

$$InitSt \ \widehat{=}\ max\_rep, counter := 4, 0$$

The *Circus* action *Send* starts the *Ping* request by sending a signal to the address *ip* via the channel $send\_data$. Then, the signal *tick* marks that one time unit is elapsed and finally the action ends by executing the *Receive* *Circus* action with the parameter set to 1, meaning that one time unit has elapsed since the request for the server *ip*.

$$Send \ \widehat{=}\ send\_data!ip \longrightarrow tick \longrightarrow Receive(1)$$

Next, the *Receive* *Circus* action, which has an input variable, *time*, is defined as an external choice between (1) a received signal from the server *ip*, and then it displays the time of the received message, or (2) a signal *tick*, meaning that time is passing with no answer from *ip* and ending with a recursion of *Receive* with the value of *time* incremented by one time unit.

$$Receive \mathrel{\widehat=} time : \mathbb{N} \bullet \left(\begin{array}{l} receive\_data?ip \longrightarrow display\_resp!(REC, time) \longrightarrow counter := counter + 1 \\ \Box tick \longrightarrow Receive(time + 1) \end{array}\right)$$

We define the behaviour of the *Ping* requests by defining the *Circus* action, *Run* which is a recursion. We formalise a recursion in *Circus* by using the constructs $\mu X \bullet P ;\ X$, where P is the action being executed. Two behaviours are expected during the recursion, which are specified in an **if** $P \longrightarrow Q \mathrel{[]} R \longrightarrow S$ **fi** predicate: if P is satisfied, then the action behaves like Q; otherwise, if R is satisfied, then it behaves like S. The behaviours expected in the *Run* process are the following. (1) If the number of requests has not been reached, $counter < max\_rep$, the *Send* *Circus* action is executed with a maximum execution time of 1000 time units , and will be timed out after reaching this interval ($\overset{1000}{\rhd}$). Then, after the timeout of the *Send* action, the *Run* *Circus* action checks if the size of the sequence *req_packets* is equal to the number of ping requests. If the size of *req_packets* is less than *counter*, then, it means that the server *ip* has not answered the request before the timeout and thus the component *req_packet* component is concatenated with another element of the sequence, which states that there was no answer from the server, and the *counter* component is incremented by one unit. However, if the size of the sequence *req_packet* is equal to *counter*, it means that there has been an answer from *ip*, and then, the *counter* component is incremented by one unit. (2) If the number of requests, *counter* has reached the maximal number of requests *max_rep*, then, a signal **Skip** ends the recursion and terminates the *Circus* action *Run*.

$$Run \mathrel{\widehat=} \mu X \bullet \left(\begin{array}{l} \mathbf{if}(counter < max\_rep) \longrightarrow \\ \left(\begin{array}{l} Send \overset{1000}{\rhd} \\ \left(\begin{array}{c} display\_resp!(TIMEOUT, time) \longrightarrow \\ counter := counter + 1 \end{array}\right) \end{array}\right) ;\ X \\ \mathrel{[]}(counter = max\_rep) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

The main action of the *Ping* *Circus* process consists in the sequential composition of the *InitSt* and *Run* *Circus* actions.

$\bullet\ InitSt ;\ Run$

end

In this example we give an overview of a *Circus* process. During the construction of out model in the next chapter, we present a few other operator like those used for interactions between *Circus* processes can be modelled using CSP operators such as parallelism and choice. Moreover, we will detail the translation from *Circus* to CSP and how we introduce the *Circus Time* constructs into CSP.

2.5 Final Considerations

We have presented in this chapter the structure of the IMA architecture, detailing its components. IMA applications are allocated within a partition, with predefined time and memory resources. Each partition can have one or more internal processes, which are not visible outside the partition boundaries. Partitions have access to the resources of its module, such as sensors and actuators, through request of the APEX services. The schedule of execution of the partitions within a module is defined in the configuration tables. Finally, the health monitor is the component of the module that controls failures and errors detected within the module and provides the correct recovery action for the errors, specified in the configuration tables.

Moreover, we have given an overview of existing approaches on certification and verification of aircraft systems. We focus on those works that apply formal methods to the verification of IMA systems. We have concluded this chapter with a survey of existing formal languages for specification of concurrent systems. By using *Circus* we are able to formalise the IMA architecture, to capturing both data and the behaviour of the architecture and also model scheduling capabilities of the architecture. This is subject of the next chapter.

Chapter 3

A *Circus* Model of the ARINC 653 Components

In this chapter we present how we formalise the three top layers of the IMA architecture using *Circus*, according to the ARINC 653 requirements. The model presented here covers general features required by the ARINC specification and provides basic services to the IMA applications in general. After producing the formal model, we translate it into CSP in order to validate it using the model checker FDR [30], and are able to animate it using ProBE [31].

In our model, we define a *Circus* process for each component of the module, illustrated as rectangular blocks in the Fig. 3.1. The communication between the *Circus* processes is made using *Circus* channels, illustrated as arrows in that figure, indicating the direction of the data communication.

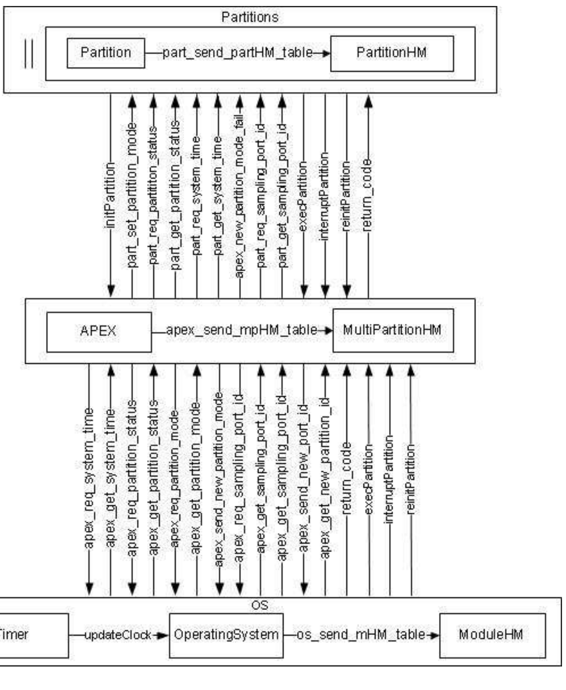

Figure 3.1: ARINC 653 - Overview of the channels used in the *Circus* model of the architecture

The figure presents the three top levels of the ARINC 653 architecture as previously presented and illustrated by the Fig. 2.1: the operating system on the bottom, the Application Executive (APEX) in the middle and the partitions on the top.

As an ARINC module can have more than one partition running within the module, we model the set of partitions as the *Circus* process *Partitions*, which is a parallel composition of each *A653_Partition* *Circus* process. We use parallelism since the partitions operate concurrently, but they must synchronise between them three times during the execution of the module: when it is switched on, with the signal *moduleInit*; at the end of the initialisation, synchronising on *moduleEndInit*; and when the module is to be switched off, with the signal *moduleEnd*. Besides those synchronisations, any other direct communication between partitions is not allowed according to the ARINC 653 specification. The communication between partitions, referred as interpartition communication, is made through the *APEX* services.

The process *Partitions* has tree inputs. The first one is the sequence of fixed attributes of each partition, defined in the configuration tables. The other two inputs are sequences of data related to the health monitor for individual partitions: the set of errors and specific recovery actions for each partition. The inputs of *Partitions* are the inputs of each of the interleaved *A653_Partition* process.

The *APEX* *Circus* process manages the services used for communication, scheduling and managing the status of the partitions running within the ARINC module. It has two inputs, related to the health monitor error list and recovery action tables in the level of the set of partitions.

The *OperatingSystem* *Circus* process has three inputs: the sequence of execution of the partitions, which contains, for each partition, its name and timing properties such as duration and period, the list of errors, and recovery actions for the health monitor in the module level.

Although each partition has a particular identifier, the paragraph of the configuration table that contains information regarding the sequence of execution in the module identifies each partition by its name. However, partitions are referred in the *APEX* services by their identifiers. For that reason, we define that each *Circus* channel used for the communication between the *APEX* and the partitions carries the identifier of the partition that is currently accessing the resources of the module. Moreover, internal operations in the operating system are used to calculate the partition identifier related to the name of the partition to be executed within the schedule.

We model the three levels of the ARINC health monitor as three *Circus* processes: *ModuleHM*, *MultiPartitionHM* and *PartitionHM*. The recovery actions related to the ARINC module are managed by the *ModuleHM* process. The recovery actions in the module level are used when the error raised is not synchronous to execution of a partition. When the error is related to a partition in execution, the *MultiPartitionHM* process analyses whether or not the recovery action is in the context of the partition or of the module itself. Finally, if the raised error impacts processes of the partition in execution, the *PartitionHM* process decides the correct recovery action for the context of error inside the partition. In this dissertation, however, we do not model the health monitor recovering actions for any of the architecture levels. This is part of our future work and is discussed in Section 4.2.

In this chapter, we will present the formalisation of the IMA architecture using *Circus*. Firstly, we present how we model the ARINC 653 types and *Circus* channels to be used in our model. Then, we introduce the *Circus* model for the partitions layer, a model for the APEX layer, and we present our model for the operating system and how we capture the scheduling capability and the temporal partitioning properties. After presenting the *Circus* model of the architecture, we present its translation into CSP and finally we conclude this chapter with the validation of the model using FDR.

3.1 Common Types and Channels

The following types are used for modelling the configuration tables and defining the types of data communicated through *Circus* channels for the $APEX$ services. We follow the ARINC 653 notation for modelling types used in our specification, by using upper-case letters and underscores to refer to types in our *Circus* model. The channels presented in this section are used to describe the execution of an IMA module. Channels used for communication between *Circus* processes of the model are presented in the description of the components of our specification in this chapter. The complete list of those channels can be found in the Appendix A.2 of this dissertation.

3.1.1 Types

The ARINC 653 document defines the types of variables used in the configuration tables and APEX services. These types are generally natural numbers, strings, and sets of constants and identifiers of memory allocated areas, which depend on the programming language used for implementation. In order to model these types, we adopt the following approach. For those types that depend on the programming language, we define a Z given set. For those types that do not depend on any programming language, like a type of natural numbers, we define a name (Z abbreviation) as specified in the ARINC 653 document. In order to model the structure of the ARINC 653 configuration tables, we use Z schemas. In this section, we provide a few examples of how we formalise the types of the ARINC module using *Circus*. Other types are presented in the Appendix A.1.

The ARINC 653 document specifies that the sequence of execution of the partitions within a module depends on the order defined in the configuration tables. Moreover, at least one partition should be allocated in the module. We can formalise these requirements using the following definition: a non-empty injective sequence of type X. The formal definition of $iseq_1[X]$ is presented below.

$$iseq_1[\,X\,] == \{\, s : \operatorname{seq} X \mid s \neq \langle\rangle \land s \in \mathbb{N} \rightarrowtail X \,\}$$

The type $DecOrHexValueType$ can have integer decimal or hexadecimal values. According to the ARINC 653 document, the definition of types in both Ada and C uses integers for numeric values. For that reason, we define the type *DecOrHexValueType* as the set of integers. The type *IdentifierValueType*, a new name for the type *DecOrHexValueType*, is the type of identifiers in the configuration table.

$$DecOrHexValueType == \mathbb{Z}$$

We define the type *NameType* for names of the partitions and ARINC 653 processes, which is type of *String*.

$$[\,String\,]$$
$$NameType == String$$

An example of the formalisation of the types used in the APEX module is the type of the time-related variables, like the elapsed time since the module was switched on. The type for these variables is *SYSTEM_TIME_TYPE*, which is defined as *DecOrHexValueType*: integer numbers with a minimum time interval of one nanosecond.

$$SYSTEM\_TIME\_TYPE == DecOrHexValueType$$

Every APEX service is designed in such a way to return informative messages stating whether the service has been successfully executed or detailing, in case of failure, the possible cause of the problem. The ARINC specification document defines *RETURN_CODE_TYPE* as the type of the return messages, and we define it in Z as a set of constants characterised by a free type.

$$\begin{aligned} RETURN\_CODE\_TYPE ::= {} & NO\_ERROR \mid NO\_ACTION \mid NOT\_AVAILABLE \\ & \mid INVALID\_PARAM \mid INVALID\_CONFIG \\ & \mid INVALID\_MODE \mid TIMED\_OUT \end{aligned}$$

There are some constants that are used in multiple types. For that, we create the type *ARINC_CONSTANTS*, which contains the set of all constants used in the definition of the ARINC 653 types, and then, we define the new types as subsets of *ARINC_CONSTANTS*.

$$\begin{aligned} ARINC\_CONSTANTS ::= {} & COLD\_START \mid WARM\_START \mid COLD\_RESTART \\ & \mid WARM\_RESTART \mid IDLE \mid NORMAL \\ & \mid ERROR\_MODE \mid IGNORE \mid SHUTDOWN \\ & \mid RESET \mid ARINC\_CONSTANTS\_NULL \end{aligned}$$

As an example, we define the possible operation modes of a partition. The possible operating modes are *NORMAL*, *IDLE*, *COLD_START* and *WARM_START* for initialisation. Moreover, according to the specification, the operation that changes the operating mode of a partition must not accept any other operating mode; it returns a message stating that the parameter is invalid if it receives a different operating mode.

In order to include that operating modes in our model, we define a subset of the

$ARINC\_CONSTANTS$, which includes the set of the possible operating modes. In Z, we specify a subset of a type, such as $ARINC\_CONSTANTS$, by using the operator '\', which excludes all constants described in the right-hand side of the operator from the set on the left-hand side.

We define the set of accepted operating modes, $OPERATING\_MODE\_TYPE$, as a subset of $ARINC\_CONSTANTS$, restricted to the $NORMAL$, $IDLE$, $COLD\_START$ and $WARM\_START$ constants, excluding the rest of the constants such as $COLD\_RESTART$, $WARM\_RESTART$ and $ERROR\_MODE$. For the initialisation of a partition, we restrict the set of operating modes to only $COLD\_START$ or $WARM\_START$. The type $OPERATING\_INIT\_MODE\_TYPE$, defined below, is also a subset of $ARINC\_CONSTANTS$.

$$\begin{aligned}
&OPERATING\_MODE\_TYPE == \\
&\qquad ARINC\_CONSTANTS \setminus \{COLD\_RESTART, WARM\_RESTART, \\
&\qquad\qquad ERROR\_MODE, IGNORE, SHUTDOWN, \\
&\qquad\qquad RESET, ARINC\_CONSTANTS\_NULL\} \\
&OPERATING\_INIT\_MODE\_TYPE == \\
&\qquad ARINC\_CONSTANTS \setminus \{IDLE, NORMAL, \\
&\qquad\qquad COLD\_RESTART, WARM\_RESTART, \\
&\qquad\qquad ERROR\_MODE, IGNORE, SHUTDOWN, \\
&\qquad\qquad RESET, ARINC\_CONSTANTS\_NULL\}
\end{aligned}$$

In the definition of the partition state, we use the $START\_CONDITION\_TYPE$, which represents the circumstances of the initialisation of the partition. Its constants defined below allow the system to know whether it was initialised normally or if it was re-initialised due to a failure. The possible start conditions are: $NORMAL\_START$, in case of a normal power-up; $PARTITION\_RESTART$, in the case of a restart using an APEX service to set the partition operating mode to either $COLD\_START$ or $WARM\_START$; HM_NORMAL_START, in the case of a recovery action performed based on a decision of the Health Monitor at module level; and $HM\_PARTITION\_RESTART$ in the case of a recovery action at partition level.

$$\begin{aligned}
&START\_CONDITION\_TYPE ::= NORMAL\_START \mid PARTITION\_RESTART \\
&\qquad\qquad \mid HM\_NORMAL\_START \mid HM\_PARTITION\_RESTART
\end{aligned}$$

We also define the lock level type used to enable or disable preemption in the partition. Partitions may have different lock levels depending on their level of criticality. The ARINC 653 specifies that lock levels are integers, and for that reason, in our model, the $LOCK\_LEVEL\_TYPE$ is modelled as a Z abbreviation of $DecOrHexValueType$.

$$LOCK\_LEVEL\_TYPE == DecOrHexValueType$$

In the next section we present the channels used for describing the execution of the IMA module.

3.1.2 Channels

The life cycle of an ARINC module is illustrated in the Fig. 3.2. In our *Circus* model, the transition between the possible states of an ARINC 653 module is communicated to the components of the module using signals over *Circus* channels, illustrated in italic characters below each transition.

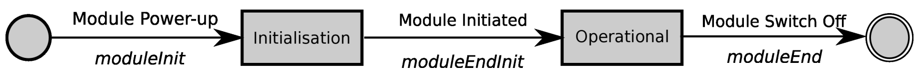

Figure 3.2: ARINC 653 - Channels for the module state transition

When the module is powered on, synchronisation on the channel *moduleStart* is used to inform the partitions that the APEX is in the initialisation state. At this point, all the partitions are initialised. When the initialisation is successfully completed, a signal on the channel *moduleEndInit* indicates to the operating system that it is to start to manage the partitions, processes and communication. That signal also informs the partitions that the module is operational and the processes inside the partitions are ready to run. Moreover, during its lifetime, a partition may be powered off. In this case, the channel *moduleEnd* causes an interruption, ending the execution of the partitions.

$$\mathbf{channel}\ \ moduleInit, moduleEndInit, moduleEnd$$

We conclude the description of the common types and channels used in our model of the IMA architecture. In the next sections we present how we model the structure of each of the layers of the IMA architecture using *Circus*.

3.2 Partitions

We model a partition as a *Circus* process, *A653_Partition*, with three parameters: a *partitionId* of type *PARTITION_ID_TYPE*, which is the identifier of a partition; the sequence of recovery actions, *partHM*, for the partition level; and the list of possible errors, *sysError*, that can be detected within the ARINC module, defined in the configuration tables. In this first model, *A653_Partition* *Circus* processes have no state, since we are still not dealing with internal ARINC 653 processes within the partitions. However, we provide here how those processes will communicate with the partition requesting to have access to the *APEX* services. In general lines, a process request to its associated partition to have access to the *APEX* services through a given *Circus* channel. Then, the partition will send that request to the *APEX*, receive the result of the request and send it back to the process through another *Circus* channel. A new model including those processes is part of our future work and is discussed in Section 4.2.

$$\begin{array}{l}\mathbf{process}\ \ A653\_Partition \mathrel{\widehat=} partitionId : PARTITION\_ID\_TYPE;\\ \qquad partHM : A653\_PartitionHMTableType;\\ \qquad\quad sysError : \mathrm{seq}_1\ A653\_ErrorIdentifierType \bullet \mathbf{begin}\end{array}$$

The first *Circus* action of the *A653_Partition* *Circus* process is *GetTime*, which obtains the current system time from the ARINC 653 operating system through a request of the *APEX* service *GET_TIME*. The action *GetTime* starts with a signal from an ARINC 653 process, *processId*, of the partition *partitionId* in execution via the channel *proc_req_system_time*. Then, it sends a signal to the *APEX* with the identifier of the partition, *partitionId*, requesting the system time of the operating system through the channel *part_req_system_time*. Then, the *APEX* responds to the request with the system time t through the channel *part_get_system_time*, which is sent back to the requesting process through the channel *proc_get_system_time*. Finally, the return code *rc* is received from the *APEX* through the channel *return_code*.

$$\begin{array}{l}
GetTime \mathrel{\widehat=} \\
\quad proc\_req\_system\_time.partitionId?processId \longrightarrow \\
\quad part\_req\_system\_time.partitionId \longrightarrow \\
\quad part\_get\_system\_time.partitionId?t \longrightarrow \\
\quad proc\_get\_system\_time.partitionId.processId!t \longrightarrow \\
\quad return\_code.partitionId?rc \longrightarrow \mathbf{Skip}
\end{array}$$

In order to change the operating mode of the partition, the *SetPartitionMode* *Circus* action is used. Similar to the *GetTime* action, it receives a signal from the executing process, *processId* with the new operating mode for that partition, *OPERATING_MODE*, and then sends it to the *APEX*, through the channel *part_req_set_partition_mode*. Then, the *APEX* responds to the request with the return code of the service, *rc* stating whether or not the *APEX* *SET_OPERATING_MODE* was successfully executed or if an error was raised.

$$\begin{array}{l}
SetPartitionMode \mathrel{\widehat=} \\
\quad proc\_set\_partition\_mode.partitionId?processId?OPERATING\_MODE \longrightarrow \\
\quad part\_set\_partition\_mode.partitionId!OPERATING\_MODE \longrightarrow \\
\quad return\_code.partitionId?rc \longrightarrow \mathbf{Skip}
\end{array}$$

A process of the partition in execution can request, through the *APEX* service *GET_PARTITION_STATUS*, the current status of a partition. The action *GetPartitionStatus* receives a request of the partition status from the process *processId* through the channel *proc_req_partition_status*, then requests it to the *APEX*, through a signal *part_req_partition_status*, which carries the identifier of the partition in execution. Then, the partition status, *st*, is received from the *APEX* through the channel *part_get_partition_status*, and the received values are sent back to the requesting process within the partition through the channel *proc_get_partition_status*. Finally, the return code, *rc*, is received from the *APEX*.

$$
\begin{array}{l}
GetPartitionStatus \mathrel{\widehat=} \\
\quad proc\_req\_partition\_status.partitionId?processId \longrightarrow \\
\quad part\_req\_partition\_status.partitionId \longrightarrow \\
\quad part\_get\_partition\_status.partitionId?st \longrightarrow \\
\quad proc\_get\_partition\_status.partitionId.processId!st \longrightarrow \\
\quad return\_code.partitionId?rc \longrightarrow \mathbf{Skip}
\end{array}
$$

The *Circus* action *GetSamplingPortId* obtains the identifier of a sampling port of the partition in execution, through request to the *APEX* service *GET_SAMPLING_PORT_ID*. The internal process in execution within a partition, *processId*, requests the identifier associated to the sampling port name, *SAMPLING_PORT_NAME*, through the channel *proc_req_sampling_port_id*. Then, the channel *part_req_sampling_port_id* sends the request to the *APEX*. Finally, the channel *part_get_sampling_port_id* receives the identifier, *spid* from the *APEX* and then sends it back to the process *processId* through the channel *proc_get_sampling_port_id*.

$$
\begin{array}{l}
GetSamplingPortId \mathrel{\widehat=} \\
\quad proc\_req\_sampling\_port\_id.partitionId?processId?SAMPLING\_PORT\_NAME \longrightarrow \\
\quad part\_req\_sampling\_port\_id.partitionId!SAMPLING\_PORT\_NAME \longrightarrow \\
\quad part\_get\_sampling\_port\_id.partitionId?spid \longrightarrow \\
\quad proc\_get\_sampling\_port\_id.partitionId!processId!spid \longrightarrow \\
\quad return\_code.partitionId?rc \longrightarrow \mathbf{Skip}
\end{array}
$$

The *Circus* action *ExecPartitionServices* starts with a signal *execPartition* from the operating system and then it starts a recursion that offers the above presented *Circus* actions, *GetTime*, *SetPartitionMode*, *GetPartitionStatus* and the *GetSamplingPortId*. The execution of the *ExecPartitionServices* may be interrupted at any time by the operating system through a signal *interruptPartition*, sent when the duration period of the execution of the partition is over.

$$
\begin{array}{l}
ExecPartitionServices \mathrel{\widehat=} \\
\quad execPartition.partitionId \longrightarrow \\
\quad \left(\begin{array}{l}
\mu X \bullet \left(\begin{array}{l} SetPartitionMode \\ \Box\ GetPartitionStatus \\ \Box\ GetSamplingPortId \\ \Box\ GetTime \end{array} \right) ;\ X \\
\triangle\ (interruptPartition.partitionId \longrightarrow \mathbf{Skip})
\end{array} \right)
\end{array}
$$

The *Circus* action *ExecPartition* describes the behaviour of an ARINC partition during the life time of a module. The action starts with a signal *initPartition* sent by the operating system, with the identifier of the partition, *partitionId*, which indicates that the partition is being initialised. After it is initialised, the partition sends the list of possible errors and the recovery actions to the partition level health monitor, through the channel *part_send_mpHM_table*. Then, it recurses offering the *Circus* action *ExecPartitionServices* in each iteration. It may be interrupted by either a signal *endPartition* indicating that the partition is being disabled or either *reinitPartition* followed by the action *ReinitPartition* indicating that the partition is being reinitialised.

$$\begin{array}{l} ExecPartition \mathrel{\widehat=} initPartition.partitionId \longrightarrow \\ \quad part\_send\_partHM\_table.partitionId!pHM!sysError \longrightarrow \\ \quad moduleEndInit \longrightarrow \\ \quad \left(\begin{array}{l} (\mu X \bullet ExecPartitionServices \ ; \ X) \\ \triangle \left(\begin{array}{l} endPartition.partitionId \longrightarrow \mathbf{Skip} \\ \Box\ reinitPartition.partitionId \longrightarrow ReinitPartition \end{array} \right) \end{array} \right) \end{array}$$

The *Circus* action *ReinitPartition* starts with a signal *partHM_restart_partition* from the Health Monitor. Then, it recurses offering the *Circus* action *ExecPartitionServices* in each iteration and may be interrupted by either a signal *endPartition* indicating that the partition is being disabled or *reinitPartition* followed by the action *ReinitPartition*.

$$\begin{array}{l} ReinitPartition \mathrel{\widehat=} partHM\_restart\_partition.partitionId \longrightarrow \\ \quad \left(\begin{array}{l} (\mu X \bullet ExecPartitionServices \ ; \ X) \\ \triangle \left(\begin{array}{l} endPartition.partitionId \longrightarrow \mathbf{Skip} \\ \Box\ reinitPartition.partitionId \longrightarrow ReinitPartition \end{array} \right) \end{array} \right) \end{array}$$

The main action of the *A653_Partition* *Circus* process describes how a partition is executed. The execution of a partition starts with a signal that the *APEX* is being initialised, through the channel *moduleInit* and then, the *Circus* action *ExecPartition* is executed until the interruption from the *APEX* through the *moduleEnd* channel indicating that the module will be switched off.

$$\bullet\ moduleInit \longrightarrow (ExecPartition \triangle\ moduleEnd \longrightarrow \mathbf{Skip})$$

end

An ARINC 653 module allows one or more partitions to run at the same time in that module. Each partition has its own health monitor, modelled here as *PartitionHM* *Circus* processes, related to each partition running under the same module. The *Circus* process *PartitionHM* is composed by a state *PartitionHMSt* which stores the values of the system errors, *sysError*, and the possible recovery actions for that partition, *pHM*.

In our model, we do not capture the behaviour of the health monitor for any of the three layers of the architecture presented in this dissertation. In the current stage of our model, the only action that occurs in the *PartitionHM* *Circus* process consists in an input channel, *part_send_partHM_table* with the values of possible errors, *se*, and recovery actions, *phm*. Then the state variables are updated with these values. At any moments, the *PartitionHM* process may be interrupted by a signal *moduleEnd* or *endPartition*.

$$
\begin{array}{l}
\textbf{process}\ \mathit{PartitionHM} \mathrel{\widehat=} \mathit{partitionId} : \mathit{PARTITION\_ID\_TYPE} \bullet \textbf{begin}\\[1em]
\quad \textbf{state}\ \mathit{PartitionHMSt} == [\mathit{pHM} : \mathit{A653\_PartitionHMTableType};\\
\qquad\qquad \mathit{sysError} : \mathrm{seq}_1\ \mathit{A653\_ErrorIdentifierType}]\\[1em]
\quad \bullet \left(\begin{array}{l} \mathit{part\_send\_partHM\_table.partitionId?phm?se} \longrightarrow\\ \mathit{pHM}, \mathit{sysError} := \mathit{phm}, \mathit{se}\end{array}\right)\\
\qquad\qquad \bigtriangleup \left(\begin{array}{l} \mathit{moduleEnd} \longrightarrow \textbf{Skip}\\ \Box\ \mathit{endPartition.partitionId} \longrightarrow \textbf{Skip}\end{array}\right)\\[1em]
\textbf{end}
\end{array}
$$

Thus, each partition, modelled as a *A653_Partition* *Circus* process is put in parallel with its health monitor, the *PartitionHM* *Circus* process, and communicates with each other via the channels that compose the *PartitionHMChannels* channel set.

$$
\begin{array}{l}
\textbf{channelset}\ \mathit{PartitionHMChannels} ==\\
\qquad\qquad \{\!|\ \mathit{part\_send\_partHM\_table}, \mathit{moduleEnd}, \mathit{endPartition}\ |\!\}
\end{array}
$$

We define the layer that contains the ARINC partitions as the *Partition_Layer* *Circus* process. It has three inputs: the parameter *partitionIds* of type $\mathbb{P}\ \mathit{PARTITION\_ID\_TYPE}$, which is a set of partition identifiers; the sequence of recovery actions, *partHM*, for the partition level; and the list of possible errors, *sysError*, that can be detected within the ARINC module. For each partition with identifier *id* in the set of *partitionIds*, the elements of the input of the *Partition_Layer* process are associated to the inputs of each *A653_Partition* *Circus* process and its related *PartitionHM* process. Each partition synchronises with the channels *moduleInit*, *moduleEndInit* and *moduleEnd*. Each ARINC partition is defined as a process *A653_Partition* *Circus* process, in parallel with its associated *PartitionHM* *Circus* process.

$$
\begin{array}{l}
\textbf{process}\ \mathit{Partitions\_Layer} \mathrel{\widehat=} \mathit{partitionIds} : \mathbb{P}\ \mathit{PARTITION\_ID\_TYPE};\\
\quad \mathit{partHM} : \mathrm{seq}_1\ \mathit{A653\_PartitionHMTableType};\\
\qquad \mathit{sysError} : \mathrm{seq}_1\ \mathit{A653\_ErrorIdentifierType}\\
\qquad \bullet \left(\begin{array}{l} \|\ \mathit{pid} : \mathit{partitionIds}\ [\![\ \{\!|\ \mathit{moduleInit}, \mathit{moduleEnd}, \mathit{moduleEndInit}\ |\!\}\ ]\!]\\ \bullet \left(\begin{array}{l} \mathit{A653\_Partition}(\mathit{pid}, \mathit{partHM}(\mathit{pid}), \mathit{sysError})\\ \qquad [\![\ \mathit{PartitionHMChannels}\ ]\!]\ \mathit{PartitionHM}(\mathit{pid})\end{array}\right)\end{array}\right)
\end{array}
$$

In the next section we present our model for the Application Executive (APEX) using *Circus*.

3.3 The APEX

The *APEX* *Circus* process is modelled with two parameters from the configuration tables: the set of possible errors that can happen within the system (*sysError*) and the multipartition level health monitor actions (*mpHM*).

$$\begin{aligned}\textbf{process}\ APEX\ \widehat{=}\ & sysError : \mathrm{seq}_1\ A653\_ErrorIdentifierType;\\ & mpHM : \mathrm{seq}_1\ A653\_MultiPartitionHMTableType \bullet \textbf{begin}\end{aligned}$$

The action *GET_TIME* is the *APEX* service that requests the current time to the operating system. The *APEX* receives the request from the partition in execution, *pid*, via the channel *part_req_system_time*, and then sends a signal to the operating system through the channel *apex_req_system_time* to indicate that the partition *pid* is requesting the system time. Then the operating system returns the time t through the channel *apex_get_system_time*, and then the *APEX* sends the time to the partition via the channel *part_get_system_time*.

$$\begin{aligned}& GET\_TIME\ \widehat{=}\ part\_req\_system\_time?pid \longrightarrow\\ &\quad apex\_req\_system\_time.pid \longrightarrow\\ &\quad apex\_get\_system\_time.pid?t \longrightarrow\\ &\quad part\_get\_system\_time.pid!t \longrightarrow\\ &\quad return\_code.pid!NO\_ERROR \longrightarrow \textbf{Skip}\end{aligned}$$

The second action is *SET_PARTITION_MODE*, through which the *APEX* process can change the operating modes of each partition. This action receives the new operating mode (*nopm*) from the partition in execution, through the channel *part_set_partition_mode* and then, it behaves according to the conditions specified for that service in the ARINC document.

If the new operating mode *nopm* does not belong to the possible operating modes, *OPERATING_MODE_TYPE*, then the channel *return_code* sends a signal that the value received is invalid. However, if the actual operating mode *opm* is *NORMAL* and the value of *nopm* is also *NORMAL*, then a signal *apex_new_partition_mode_fail* is sent and the *return_code* channel sends the message *NO_ACTION* stating that the mode was not updated. Another condition is if the value of *nopm* leads the operating mode of the partition to *IDLE* or any of the *OPERATING_INIT_MODE_TYPE*, the channel *apex_req_new_partition_mode* sends the *nopm* value to the operating system and the *return_code* channel sends a signal with a *NO_ERROR* value. Finally, in case the current operating mode *nopm* is not *NORMAL* and the new value of *nopm* is *NORMAL*, then the channel *apex_req_new_partition_mode* sends the *nopm* value to the operating system and the *return_code* channel sends a signal with a *NO_ERROR* value.

$$
\begin{array}{l}
SET\_PARTITION\_MODE \mathrel{\widehat=} \\
\quad part\_set\_partition\_mode?pid?nopm \longrightarrow \\
\quad apex\_req\_partition\_mode.pid \longrightarrow \\
\quad apex\_get\_partition\_mode.pid?opm \longrightarrow \\
\qquad \left(\begin{array}{l}
\mathbf{if}\ (nopm \notin OPERATING\_MODE\_TYPE) \longrightarrow \\
\qquad return\_code.pid!INVALID\_PARAM \longrightarrow \mathbf{Skip} \\
[]\ (opm = NORMAL \land nopm = NORMAL) \longrightarrow \\
\qquad apex\_new\_partition\_mode\_fail.pid \longrightarrow \\
\qquad return\_code.pid!NO\_ACTION \longrightarrow \mathbf{Skip} \\
[]\ (opm = COLD\_START \land nopm = WARM\_START) \longrightarrow \\
\qquad apex\_new\_partition\_mode\_fail.pid \longrightarrow \\
\qquad return\_code.pid!INVALID\_MODE \longrightarrow \mathbf{Skip} \\
[]\ \left(\begin{array}{l} nopm = IDLE \\ \lor\ nopm \in OPERATING\_INIT\_MODE\_TYPE \end{array}\right) \longrightarrow \\
\qquad apex\_req\_new\_partition\_mode.pid!nopm \longrightarrow \\
\qquad return\_code.pid!NO\_ERROR \longrightarrow \mathbf{Skip} \\
[]\ (opm \neq NORMAL \land nopm = NORMAL) \longrightarrow \\
\qquad apex\_req\_new\_partition\_mode.pid!nopm \longrightarrow \\
\qquad return\_code.pid!NO\_ERROR \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

One of the actions of the *APEX* process used to manage partitions is the service *GET_PARTITION_STATUS*. Partitions must not have direct access to the properties defined in the configuration tables. This information is managed by the operating system and is provided to the partition through a request via the *GET_PARTITION_STATUS* service. First, the identifier of the current partition is received from the partition through the channel *part_req_partition_status*, and then it is sent via the channel *apex_req_partition_status* to the operating system. The operating system returns the current status of the partition, *cp*, through the channel *apex_get_partition_status*, and finally the channel *part_get_partition_status* sends back to the partition its current status.

$$
\begin{array}{l}
GET\_PARTITION\_STATUS \mathrel{\widehat=} \\
\quad part\_req\_partition\_status?pid \longrightarrow \\
\quad apex\_req\_partition\_status.pid \longrightarrow \\
\quad apex\_get\_partition\_status.pid?cp \longrightarrow \\
\quad part\_get\_partition\_status.pid!cp \longrightarrow \mathbf{Skip}
\end{array}
$$

Another service of the *APEX* that gets the identifier of a sampling port of the current partition, *pid*, is modelled as the *Circus* action *GET_SAMPLING_PORT_ID*. The partition sends the name of the requested port, *spn*, through the channel *part_req_sampling_port_id* and then the name of the port is sent to the operating system via the channel *apex_req_sampling_port_id*. The operating system responds through the channel *apex_get_sampling_port_id* with a number, *spid*, that is verified: if the received identifier is equal to -1, then that sampling port identifier is sent to the partition along with an error of invalid configuration; otherwise, the sampling port identifier is sent to the partition, but no error is reported.

$$
\begin{array}{l}
\mathit{GET\_SAMPLING\_PORT\_ID} \mathrel{\widehat=} \\
\quad \mathit{part\_req\_sampling\_port\_id}?\mathit{pid}?\mathit{spn} \longrightarrow \\
\quad \mathit{apex\_req\_sampling\_port\_id}.\mathit{pid}!\mathit{spn} \longrightarrow \\
\quad \mathit{apex\_get\_sampling\_port\_id}.\mathit{pid}?\mathit{spid} \longrightarrow \\
\quad\quad \left(\begin{array}{l}
\mathbf{if}\ (\mathit{spid} = \text{-}\,1) \longrightarrow \\
\quad\quad \mathit{part\_get\_sampling\_port\_id}.\mathit{pid}!\mathit{spid} \longrightarrow \\
\quad\quad \mathit{return\_code}.\mathit{pid}!\mathit{INVALID\_CONFIG} \longrightarrow \mathbf{Skip} \\
[\!]\ (\mathit{spid} \neq \text{-}\,1) \longrightarrow \\
\quad\quad \mathit{part\_get\_sampling\_port\_id}.\mathit{pid}!\mathit{spid} \longrightarrow \\
\quad\quad \mathit{return\_code}.\mathit{pid}!\mathit{NO\_ERROR} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

The collection of *APEX* services are defined by the *ExecApexServices* action. The action receives as input through the channel *execPartition* the identifier of the partition in execution, *pid*, and then recurses offering the actions defined above, *GET_TIME*, *SET_PARTITION_MODE*, *GET_PARTITION_STATUS*, and *GET_SAMPLING_PORT_ID*. At anytime, the action may be interrupted by a signal from the operating system, *interruptPartition*, which is sent when the period of execution of that partition is over, or a signal *endPartition* in order to disable the partition.

$$
\begin{array}{l}
\mathit{ExecApexServices} \mathrel{\widehat=} \mathit{execPartition}?\mathit{pid} \longrightarrow \\
\quad \left(\begin{array}{l}
\mu X \bullet \left(\begin{array}{l}
\mathit{GET\_TIME} \\
\Box\ \mathit{SET\_PARTITION\_MODE} \\
\Box\ \mathit{GET\_PARTITION\_STATUS} \\
\Box\ \mathit{GET\_SAMPLING\_PORT\_ID}
\end{array}\right) ;\ X \\
\triangle \left(\begin{array}{l}
\mathit{interruptPartition}.\mathit{pid} \longrightarrow \mathbf{Skip} \\
\Box\ \mathit{endPartition}.\mathit{pid} \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}\right)
\end{array}
$$

The *Circus* action *ExecApex* allows the partition in execution to have access to those services presented above during the allowed time for its execution. The *APEX* process starts the *ExecApex* sending the set of errors and recovery actions to the Multi-Partition health monitor, through the channel *apex_send_mpHM_table*, and then, a signal *moduleEndInit* is sent, informing the partitions allocated within the module that it is running and the *APEX* services are currently available to the partitions. Then it recurses the *ExecApexServices* *Circus* action.

$$
\begin{array}{l}
\mathit{ExecApex} \mathrel{\widehat=} \mathit{apex\_send\_mpHM\_table}!\mathit{mpHM}!\mathit{sysError} \longrightarrow \\
\quad \mathit{moduleEndInit} \longrightarrow (\mu X \bullet \mathit{ExecApexServices}\ ;\ X)
\end{array}
$$

The $APEX$ is initialised with a signal from the operating system, via the channel *moduleInit*, indicating that the module is being initialised. Then, the *ExecApex* is executed during the lifetime of the $APEX$ *Circus* process until it receives a signal to terminate the process via the channel *moduleEnd*.

$$\bullet\ moduleInit \longrightarrow (ExecApex \bigtriangleup moduleEnd \longrightarrow \mathbf{Skip})$$

end

We present here the structure of the *MultiPartitionHM* *Circus* process. It behaves as follows: it receives the list of errors and recovery actions from the $APEX$ through the channel *apex_send_mpHM_table* and then, assign these values to the state variables *mpHM* and *sysError*. This model does not cover how the health monitor behaves according the list of errors, *sysError*, and the recovery actions, *mpHM*, in the multipartition level. This is part of our plans for future work and is discussed in the Section 4.2.

$$\begin{array}{l}
\mathbf{process}\ MultiPartitionHM \mathrel{\widehat=} \mathbf{begin} \\[1em]
\quad \mathbf{state}\ MultiPartitionHMSt == \\
\qquad\qquad [mpHM : \mathrm{seq}_1\ A653\_MultiPartitionHMTableType; \\
\qquad\qquad\quad sysError : \mathrm{seq}_1\ A653\_ErrorIdentifierType] \\
\quad \bullet\ apex\_send\_mpHM\_table?mp?se \longrightarrow \\
\qquad\qquad mpHM, sysError := mp, se \\
\qquad\qquad\quad \bigtriangleup moduleEnd \longrightarrow \mathbf{Skip} \\[1em]
\mathbf{end}
\end{array}$$

We define below the *Circus* channel set *MultiPartitionHMChannels* for communication between the $APEX$ and the *MultiPartitionHM* *Circus* process.

$$\mathbf{channelset}\ MultiPartitionHMChannels == \{\!|\ apex\_send\_mpHM\_table, moduleEnd\ |\!\}$$

Finally, we define the *APEX_Layer*, according to the Fig. 3.1, with the parallelism between the $APEX$ *Circus* process and the *MultiPartitionHM* *Circus* process, communicating through the channels that compose the *MultiPartitionHMChannel*.

$$\begin{array}{l}
\mathbf{process}\ APEX\_Layer \mathrel{\widehat=} mpHM : \mathrm{seq}_1\ A653\_MultiPartitionHMTableType; \\
\qquad sysError : \mathrm{seq}_1\ A653\_ErrorIdentifierType \\
\qquad\qquad \bullet \left(\begin{array}{l} APEX(sysError, mpHM) \\ \qquad [\![MultiPartitionHMChannels]\!]\ MultiPartitionHM \end{array} \right)
\end{array}$$

We have presented here the structure of $APEX$ layer of the IMA architecture. In the next section we present how we model the operating system level of the architecture using *Circus*.

3.4 The Operating System

As previously mentioned in this document, none of the partitions running within an ARINC 653 module has access to its attributes, such as port names or timing restrictions. This information is stored by the operating system. In our model, the content of the configuration tables is provided to the operating system *Circus* process. We then need to model the data structure of the configuration tables for an ARINC module based on information from the ARINC 653 document, as illustrated by Fig. 2.4.

3.4.1 Configuration tables

The configuration tables consist of a set of data which contain information regarding the components of an ARINC module, such as the requirements for each partition, schedule for execution, and health monitoring. In Z, we then model a schema *Module* with four components: the module *Name*, a set of *Partitions* to run within the partition, the *Schedules* of execution of the partitions, and a *HealthMonitor* with the list of errors and recovery actions for that module. Each of these components is presented in the sequel.

A *Partition* is composed by four components modelled as Z schemas: *A653_PartitionBaseType*, *A653_PartitionPeriodicityType*, *A653_MemoryRegionType* and *PortBaseType*. For instance, the *A653_PartitionBaseType* stores information about the *Identifier* and the *Name* of the partition, as illustrated below.

$$\begin{array}{|l}
\textit{A653\_PartitionBaseType} \\ \hline
\textit{Identifier} : \textit{PARTITION\_ID\_TYPE} \\
\textit{Name} : \textit{NameType} \\ \hline
\end{array}$$

A *Partition* is composed of four components: a *PartitionDefinition* of type *A653_PartitionBaseType*[1], containing its name and identifier; a *PartitionPeriodicity* of type *A653_PartitionPeriodicityType*, that contains records of the period and duration of its execution; a *MemoryRegions* component of type $iseq_1[A653\_MemoryRegionType]$, which is a non-empty injective sequence of memory regions allocated to that partition; and a *PartitionPorts*, which is a non-empty injective sequence of *PartitionPort* for inter-partition communication. In the *A653_Partition* schema, the use of non-empty sequences reflects the fact that there must be at least one memory region and one port associated with a partition.

$$\begin{array}{|l}
\textit{Partition} \\ \hline
\textit{PartitionDefinition} : \textit{A653\_PartitionBaseType} \\
\textit{PartitionPeriodicity} : \textit{A653\_PartitionPeriodicityType} \\
\textit{MemoryRegions} : iseq_1[\textit{A653\_MemoryRegionType}] \\
\textit{PartitionPorts} : iseq_1[\textit{PartitionPort}] \\ \hline
\end{array}$$

[1]The entire specification of the components of the configuration tables can be found in the Appendix A

As the ARINC 653 module supports multiple partitions, the type *PartitionsType* is specified a non-empty injective sequence of partitions ($iseq_1[\,Partition\,]$): there must be at least one partition allocated in that ARINC 653 module.

$$PartitionsType == iseq_1[Partition]$$

In order to model the operating system of an ARINC module, we need to define a way to identify the partitions that have access to the module resources such as processor and sensors. As presented in Fig. 2.3 in Section 2.1.2, the order of execution of the partitions within a major time frame is defined by the *Name* of the partition, *Duration* of execution, the *Offset* between the beginning of the major time frame and the execution of such partition, and whether or not the process is periodic or aperiodic. These properties are used in the scheduler section of the configuration tables and are defined as the *A653_PartititonTimeWindowType* schema below.

$$\begin{array}{l}
\textit{A653\_PartitionTimeWindowType} \\
\hline
PartitionNameRef : NameType; \\
Duration, Offset : DecOrHexValueType; \\
PeriodicProcessingStart : Boolean \\
\hline
\end{array}$$

The schedule for execution of the partitions within an ARINC 653 module, *ScheduleType*, is defined as a non-empty injective sequence of *A653_PartitionTimeWindowType*.

$$ScheduleType == iseq_1[A653\_PartitionTimeWindowType]$$

The *HealthMonitoringType* component contains records of the set of possible errors, *SystemErrors*, and recovery actions for these errors. The recovery actions are split in three levels: *ModuleHM* contains records of the possible recovery actions for errors that affects the module itself; *MultiPartitionHM* allows the health monitor to decide whether the error raised will affect only one partition, more than one partition or the entire module; and *PartitionHM* stores the recovery actions to be applied only within the boundaries of the partition in execution.

$$\begin{array}{l}
\textit{HealthMonitoringType} \\
\hline
SystemErrors : iseq_1[A653\_ErrorIdentifierType] \\
ModuleHM : iseq_1[\,A653\_ModuleHMTableType\,] \\
MultiPartitionHM : iseq_1[\,A653\_MultiPartitionHMTableType\,] \\
PartitionHM : iseq_1[\,A653\_PartitionHMTableType\,] \\
\hline
\end{array}$$

We conclude the formal structure of the configuration tables with the schema *Module*. It contains records of the presented types: *Name* of the module; the set of *Partitions* that are currently being executed within the ARINC 653 module as well as the *Schedules* of execution of the partitions, and the *HealthMonitoring* with the list of errors and recovery actions.

$$\begin{array}{|l}
\textit{Module} \\
\hline
\textit{Name} : \textit{NameType} \\
\textit{Partitions} : \textit{PartitionsType} \\
\textit{Schedules} : \textit{ScheduleType} \\
\textit{HealthMonitoring} : \textit{HealthMonitoringType} \\
\hline
\end{array}$$

Each partition has a set of attributes whose values are used by the operating system to control and maintain each partition's operation. These attributes are returned to the partition through request to the *APEX* service *GET_PARTITION_STATUS*. We define the *PartitionVariables* schema to include the set of attributes of a partition: its *OPERATING_MODE*, stating whether it is initialising, running, or idle; a *START_CONDITION*; and the *LOCK_LEVEL* for process preemption.

$$\begin{array}{|l}
\textit{PartitionVariables} \\
\hline
\textit{OPERATING\_MODE} : \textit{OPERATING\_MODE\_TYPE} \\
\textit{START\_CONDITION} : \textit{START\_CONDITION\_TYPE} \\
\textit{LOCK\_LEVEL} : \textit{LOCK\_LEVEL\_TYPE} \\
\hline
\end{array}$$

We also model the *PARTITION_STATUS_TYPE*, which is used in the *APEX* service that requests the status of the partition that is currently being executed. The service returns to the partition all its information, such as the *Identifier* of the partition, its *Period* and *Duration* of execution, and also the values of the variable attributes of that partition.

$$\begin{array}{|l}
\textit{PARTITION\_STATUS\_TYPE} \\
\hline
\textit{Identifier} : \textit{PARTITION\_ID\_TYPE} \\
\textit{Period} : \textit{SYSTEM\_TIME\_TYPE} \\
\textit{Duration} : \textit{SYSTEM\_TIME\_TYPE} \\
\textit{PartitionVariables} \\
\hline
\end{array}$$

In the next section we present our *Circus* model for the operating system of the IMA architecture.

3.4.2 *Circus* process

We formalise the ARINC 653 operating system with a *Circus* process *OperatingSystem*. The process has a single input, *module* of type *Module*, presented above.

$$\textbf{process}\ OperatingSystem \mathrel{\widehat=} module : Module \bullet \textbf{begin}$$

The state of the *OperatingSystem* *Circus* process comprises the following components: a non-empty injective sequence of *PartitionVariables*; the *major_time_frame* of the execution of the partitions; the current time in the system, *system_time*; the *current_partition* in execution within the module according to the *Schedules*; the identifier of the partition in execution, *current_partition_id*; and the set of sampling and queuing ports currently assigned within the module. The only invariant of the state is that the number of *partitions_variables* is equal to the number of *Partitions* allowed to execute within the module.

$$
\begin{array}{|l}
\multicolumn{1}{l}{OSSt} \\ \hline
partitions\_variables : \mathrm{iseq}_1[PartitionVariables] \\
major\_time\_frame : SYSTEM\_TIME\_TYPE; \\
system\_time : SYSTEM\_TIME\_TYPE \\
current\_partition : A653\_PartitionTimeWindowType \\
current\_partition\_id : PARTITION\_ID\_TYPE \\
sampling\_ports : (SAMPLING\_PORT\_ID\_TYPE \pfun \\
\qquad\qquad\qquad SAMPLING\_PORT\_NAME\_TYPE) \\
queuing\_ports : (QUEUING\_PORT\_ID\_TYPE \pfun \\
\qquad\qquad\qquad QUEUING\_PORT\_NAME\_TYPE) \\ \hline
\# partitions\_variables = \# module.Partitions \\ \hline
\end{array}
$$

$$\textbf{state}\ OSSt$$

The *Circus* action *InitPartition* initialises each partition that is going to execute in the module by sending a signal through the channel *initPartition* with the identifier of each partition, *pid*. In *Circus*, we model the interleaving between the signals sent to each of the partitions, *pids*, allowed to execute within the ARINC 653 module.

$$InitPartition \mathrel{\widehat=} \interleave\ pid : (1 \mathinner{..} (\# module.Partitions)) \bullet initPartition.pid \longrightarrow \mathbf{Skip}$$

In order to calculate the major time frame, we define the *Circus* action *MajorTimeFrame* modelled as a Z schema. We first declare that the state is being updated ($\Delta OSSt$) and we declare the variable *getMajorTimeFrame* of type *PartitionsType*. Then we introduce three predicates: firstly, the variable *getMajorTimeFrame* contains the same elements of *module.Partitions*; secondly, *getMajorTimeFrame* is ordered by the *Period* in decreasing order; and the new value of *major_time_frame* is updated with the longest period between the executions of a partition, which is the *period* of the first element of *getMajorTimeFrame*; Finally, the predicate $\theta(OSSt \setminus \{major\_time\_frame\})' = \theta(OSSt \setminus \{major\_time\_frame\})$ denotes that besides *major_time_frame*, the rest of the state variables remain unchanged after the execution of *MajorTimeFrame*.

$$
\begin{array}{l}
\textit{MajorTimeFrame} \\
\hline
\Delta OSSt \\
getMajorTimeFrame : PartitionsType \\
\hline
\forall x : \operatorname{ran} module.Partitions \\
\quad \bullet \# (module.Partitions \rhd \{x\}) = \# (getMajorTimeFrame \rhd \{x\}) \\
\forall p1, p2 : \operatorname{ran} getMajorTimeFrame \\
\quad \bullet p1.PartitionPeriodicity.Period > p2.PartitionPeriodicity.Period \\
major\_time\_frame' = (getMajorTimeFrame(1)).PartitionPeriodicity.Period \\
\theta(OSSt \setminus (major\_time\_frame))' = \theta(OSSt \setminus (major\_time\_frame)) \\
\hline
\end{array}
$$

We model the *Circus* action *NextPartition* in order to update the current partition and its identifier. The *current_partition* is assigned with the first element of the sequence *module.Schedules* restricted to those partitions where its *Offset* of the is greater than or equal to the modulo of *system_time*, i.e., the next partition to be executed within the schedule and the *major_time_frame*. Moreover, the new value of the *current_partition_id* state component is assigned with the identifier of the partition where the *Name* is equal to the *PartitionNameRef* of the *current_partition*. Only the variables *current_partition_id* and *current_partition* are updated after the execution of *NextPartition*.

$$
\begin{array}{l}
\textit{NextPartition} \\
\hline
\Delta OSSt \\
\hline
current\_partition' = head\,(module.Schedules \upharpoonright \\
\qquad\qquad \{p : A653\_PartitionTimeWindowType \mid \\
\qquad\qquad\qquad (system\_time \bmod major\_time\_frame) \leq p.Offset\}) \\
(module.Partitions(current\_partition\_id')).PartitionDefinition.Name = \\
\qquad\qquad\qquad (current\_partition').PartitionNameRef \\
\theta(OSSt \setminus (current\_partition\_id, current\_partition))' = \\
\quad \theta(OSSt \setminus (current\_partition\_id, current\_partition)) \\
\hline
\end{array}
$$

The action *UpdateSystemTime* receives a signal from the *Timer* *Circus* process with the actual value of the system time and then assigns that value to the *system_time* state component.

$$UpdateSystemTime \ \widehat{=}\ updateClock?x \longrightarrow system\_time := x$$

The *Circus* action $OSGetTime$ responds to the $APEX$ service GET_TIME, where the partition in execution requests the current time of the module. When requested, the $APEX$ sends a signal $apex\_req\_system\_time$ with the identifier of that partition in execution, then the action $UpdateSystemTime$ updates the $system\_time$ variable and finally, the current value of the $system\_time$ component is sent back to the $APEX$ through the channel $apex\_get\_system\_time$.

$$\begin{array}{l} OSGetTime \mathrel{\widehat=} apex\_req\_system\_time.current\_partition\_id \longrightarrow UpdateSystemTime; \\ \qquad apex\_get\_system\_time.current\_partition\_id!system\_time \longrightarrow \mathbf{Skip} \end{array}$$

In order to respond to the $APEX$ service $GET\_PARTITION\_STATUS$, the *Circus* action $OSGetPartitionStatus$ is used to obtain the status of the requesting partition. First, a signal
$apex\_req\_partition\_status$ is received from the APEX. Then the auxiliary action $PartitionStatus$ collects the data of the partition and stores it in the variable p of type $PARTITION\_STATUS\_TYPE$, without changing the state ($\Xi OSSt$). Finally, the channel $apex\_get\_partition\_status$ is used to send p back to the $APEX$ with the values of the current status of the requesting partition.

$$\begin{array}{|l} \multicolumn{1}{l}{PartitionStatus} \\ \hline \Xi OSSt \\ p! : PARTITION\_STATUS\_TYPE \\ \hline (p!).Identifier = \\ \quad (module.Partitions(current\_partition\_id)).PartitionDefinition.Identifier \\ (p!).Period = \\ \quad (module.Partitions(current\_partition\_id)).PartitionPeriodicity.Period \\ (p!).Duration = \\ \quad (module.Partitions(current\_partition\_id)).PartitionPeriodicity.Duration \\ (p!).LOCK\_LEVEL = \\ \quad (partitions\_variables(current\_partition\_id)).LOCK\_LEVEL \\ (p!).OPERATING\_MODE = \\ \quad (partitions\_variables(current\_partition\_id)).OPERATING\_MODE \\ (p!).START\_CONDITION = \\ \quad (partitions\_variables(current\_partition\_id)).START\_CONDITION \\ \hline \end{array}$$

$$\begin{array}{l} OSGetPartitionStatus \mathrel{\widehat=} \mathbf{var}\, p : PARTITION\_STATUS\_TYPE \bullet \\ \quad apex\_req\_partition\_status.current\_partition\_id \longrightarrow \big(PartitionStatus\big); \\ \qquad apex\_get\_partition\_status.current\_partition\_id!p \longrightarrow \mathbf{Skip} \end{array}$$

The *Circus* action $SetPMode$, below, is used in the $OSSetPartitionMode$ action. It updates the component $OPERATING\_MODE$ of the partition with identifier $current\_partition\_id$

within the sequence *partitions_variables*, with the new operating mode *nopm* received from the *APEX* in the action *OSSetPartitionMode*. The input variable *nopm*? is used to update the value of the *OPERATING_MODE* variable. We use the predicate *let*... • in order to assign new values to a variable *o* of type *PARTITION_VARIABLES*. Then the override operator $\oplus$ is used to replace the existing values in the of the sequence *partitions_variables* in the position *current_partition_id* to the new values assigned in *o*. We also state that only *partitions_variables* is updated in *OSSt*.

$$\begin{array}{|l}
\textit{SetPMode} \\
\hline
\Delta \textit{OSSt} \\
\textit{nopm}? : \textit{OPERATING\_MODE\_TYPE} \\
\hline
\textit{partitions\_variables}' = \\
(\textbf{let}\ o == \langle\!| \textit{OPERATING\_MODE} == \textit{nopm}?, \\
\textit{START\_CONDITION} == \\
\quad (\textit{partitions\_variables}(\textit{current\_partition\_id})).\textit{START\_CONDITION}, \\
\textit{LOCK\_LEVEL} == \\
\quad (\textit{partitions\_variables}(\textit{current\_partition\_id})).\textit{LOCK\_LEVEL} |\!\rangle \\
\quad \bullet \textit{partitions\_variables} \oplus \{\textit{current\_partition\_id} \mapsto o\}) \\
\theta(\textit{OSSt} \setminus (\textit{current\_partition\_id}, \textit{current\_partition}, \textit{partitions\_variables}))' = \\
\quad \theta(\textit{OSSt} \setminus (\textit{current\_partition\_id}, \textit{current\_partition}, \textit{partitions\_variables})) \\
\hline
\end{array}$$

The operating system responds to the *APEX* service *SET_PARTITION_MODE*, which requests to change the operating mode of the partition in execution, via the *Circus* action *OSSetPartitionMode*. A signal *apex_req_partition_mode* is received from the *APEX*, then, the operating system returns the current operating mode of the partition in execution, *opm*, through the channel *apex_get_partition_mode*.

$$\begin{array}{l}
\textit{OSSetPartitionMode} \mathrel{\widehat=} \textbf{var}\ \textit{opm} : \textit{OPERATING\_MODE\_TYPE} \bullet \\
\quad \textit{apex\_req\_partition\_mode}.\textit{current\_partition\_id} \longrightarrow \\
\quad \textit{apex\_get\_partition\_mode}.\textit{current\_partition\_id}!(\\
\quad (\textit{PartitionsVariables}(\textit{current\_partition\_id})).\textit{OPERATING\_MODE}) \longrightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\textit{apex\_req\_new\_partition\_mode}.\textit{current\_partition\_id}?\textit{nopm} \longrightarrow \\
\textbf{if}(\textit{nopm} = \textit{IDLE}) \longrightarrow \\
\quad \textit{endPartition}.\textit{current\_partition\_id} \longrightarrow \big(\textit{SetPMode}\big) \\
[\!] \ (\textit{nopm} \in \textit{OPERATING\_INIT\_MODE\_TYPE}) \longrightarrow \\
\quad \textit{reinitPartition}.\textit{current\_partition\_id} \longrightarrow \big(\textit{SetPMode}\big) \\
[\!] \ (\textit{nopm} = \textit{NORMAL}) \longrightarrow \big(\textit{SetPMode}\big) \\
\textbf{fi}
\end{array}\right) \\
\Box\ \textit{apex\_new\_partition\_mode\_fail}?\textit{current\_partition\_id} \longrightarrow \textbf{Skip}
\end{array}\right)
\end{array}$$

The *APEX* *Circus* process determines whether or not the new operating mode that the partition is requesting to change is acceptable. If the new operating mode is accepted by the conditions of the *APEX* service, then, the new operating mode, *nopm* is received from the *APEX* through the channel *apex_req_new_partition_mode*.

The action verifies whether or not the partition must be disabled or reinitialised. If the new mode of that partition is *IDLE*, then a signal *endPartition* is sent to the *current_partition*. Otherwise if the new mode belongs to the set *OPERATING_INIT_MODE_TYPE*, then, a signal *reinitPartition* is sent to the *current_partition*. Otherwise, if the new mode is *NORMAL*, the auxiliary action *SetPMode* assigns the new operating mode to the attributes of the partition in execution. However, if the new mode is not accepted by the *APEX*, it sends to the operating system a signal *apex_new_partition_mode_fail* and no action is taken.

The *Circus* action *OSGetSamplingPortId* responds to the request of the *APEX* service *GET_SAMPLING_PORT_ID* with the identifier of a sampling port. The *APEX* sends the request with the partition name *spn* and then the operating system does the following: if *spn* is in the range of the set of *sampling_ports*, then, the channel *apex_get_sampling_port_id* returns the identifier that corresponds to the name of the requested sampling port, and the channel *return_code* informs the *APEX* that the action was concluded with no error; otherwise, if *spn* is not in the range of *sampling_ports*, the channel *apex_get_sampling_port_id* returns - 1, used to indicate that no port was found with that name and then, *return_code* returns a message that the requested configuration is invalid.

$$
\begin{array}{l}
OSGetSamplingPortId \mathrel{\widehat=} \\
\quad apex\_req\_sampling\_port\_id.current\_partition\_id?spn \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}(spn \in \operatorname{ran} sampling\_ports) \longrightarrow \\
\quad apex\_get\_sampling\_port\_id.current\_partition\_id!(\mu \\
\qquad\qquad x : (\operatorname{dom}(sampling\_ports \rhd \{spn\})) \bullet x) \longrightarrow \\
\quad return\_code.current\_partition\_id!NO\_ERROR \longrightarrow \mathbf{Skip} \\
[\!]\ (spn \notin \operatorname{ran} sampling\_ports) \longrightarrow \\
\quad apex\_get\_sampling\_port\_id.current\_partition\_id!(-1) \longrightarrow \\
\quad return\_code.current\_partition\_id!INVALID\_CONFIG \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

On the operating system side, the set of the actions that responds to requests of the *APEX* services is defined as the *Circus* action *ExecOSServices*, which is the recursion of the external choice between the above defined actions, such as *OSGetTime* and *OSGetPartitionStatus*.

$$
\begin{array}{l}
ExecOSServices \mathrel{\widehat=} \\
\quad execPartition.current\_partition\_id \longrightarrow \\
\quad \left(\mu X \bullet \left(\begin{array}{l}
OSGetTime \\
\Box\ OSGetPartitionStatus \\
\Box\ OSSetPartitionMode \\
\Box\ OSGetSamplingPortId
\end{array}\right) ;\ X\right)
\end{array}
$$

The action *ExecOS* is executed in the main action during the execution of the *OperatingSystem* *Circus* process. It starts by calculating the *major_time_frame* with the action *MajorTimeFrame*, then it recurses as follows: at each iteration, the action *UpdateSystemTime* updates the current time within the module, then *NextPartition* calculates the *next_partition* and its identifier, *next_partition_id*. If the *Offset* of the next partition to be executed is higher than the modulo of *system_time* and *major_time_frame*, it means that the operating system must wait and no service is available to the ARINC partitions. Otherwise, the *Circus* action *ExecOS* is executed within the allowed *Duration* interval of execution of the *current_partition*. When the *Duration* is reached, a *Circus* timeout is triggered with an interrupt signal by the channel *interruptPartition* for that partition in execution and then the action recurses.

$$
\begin{array}{l}
ExecOS \mathrel{\widehat=} \mathbf{var}\, getMajorTimeFrame : PartitionsType \bullet \big(MajorTimeFrame\big); \\
\left(\begin{array}{l}
\mu X \bullet UpdateSystemTime \;;\; \big(NextPartition\big); \\
\left(\begin{array}{l}
\mathbf{if} \left(\begin{array}{l} current\_partition.Offset > \\ \qquad (system\_time \bmod major\_time\_frame) \end{array}\right) \longrightarrow \\
\qquad \mathbf{wait} \left(\begin{array}{l} current\_partition.Offset - \\ \qquad (system\_time \bmod major\_time\_frame) \end{array}\right) ;\; X \\
\square \left(\begin{array}{l} current\_partition.Offset = \\ \qquad (system\_time \bmod major\_time\_frame) \end{array}\right) \longrightarrow \\
\qquad \left(\begin{array}{l} ExecOSServices \overset{current\_partition.Duration}{\rhd} \\ \qquad interruptPartition.current\_partition\_id \;;\; X \end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}
$$

The *OperatingSystem* *Circus* process starts with a signal *moduleInit*, then it sends the list of errors and recovery actions to the *ModuleHM* through the channel *os_send_modHM_table*. Afterwards, then it executes *ExecOS*. At any time, the execution of *ExecOS* may be interrupted by a signal *moduleEnd*, which means that the module is going to be switched off.

$$
\begin{array}{l}
\bullet\; moduleInit \longrightarrow \\
\qquad os\_send\_modHM\_table!(module.HealthMonitoring.ModuleHM)!(\\
\qquad\qquad module.HealthMonitoring.SystemErrors) \longrightarrow (ExecOS \\
\qquad\qquad\quad \triangle\; moduleEnd \longrightarrow \mathbf{Skip})
\end{array}
$$

end

We conclude here the model of the operating system of the IMA architecture. We present next the construction of the timer used to represent time lapse within the IMA module.

3.4.3 The System Timer

In order to calculate the elapsed time since when the module was switched on, we model the *Circus* process *Timer*.

$\mathbf{process}\; Timer \mathrel{\widehat=} \mathbf{begin}$

We define that the *Timer* state, *TimerSt*, is composed by a single component, *clock* that contains records of the current system time.

$$\textbf{state}\ \ TimerSt == [clock : SYSTEM\_TIME\_TYPE]$$

The main action *Counter* recurses by offering the interleaving of two actions: on the left hand side, at each iteration, the system waits one nanosecond ($\textbf{wait}(1)$) and then, the clock is incremented by one nanosecond; on the right hand side, the *updateClock* channel offers the current time within the module, *clock*, for one nanosecond and then a *Circus* timeout is triggered and the action recurses again.

$$Counter \mathrel{\widehat=} \left(\begin{array}{l} \mu X \bullet \\ \left(\begin{array}{l} (\textbf{wait}(1)\ ;\ clock := clock + 1) \\ ||| \left((\mu\, Y \bullet updateClock!clock \longrightarrow Y) \overset{1}{\rhd} \textbf{Skip} \right) \end{array} \right) ;\ X \end{array} \right)$$

The *Timer* process starts with a signal *moduleInit*, then it executes *Counter* until the interruption by a signal *moduleEnd*.

$$\bullet\ moduleInit \longrightarrow (Counter \ \triangle\ moduleEnd \longrightarrow \textbf{Skip})$$

end

In the next section we introduce the model of the health monitor for the IMA module.

3.4.4 The Module Level Health Monitor

Similarly to the *MultiPartitionHM* *Circus* process, we model the *ModuleHM* *Circus* process that recovers the faults and failures within the module level. At the current stage of our work, the *ModuleHM* *Circus* process only receives the list of errors and recovery actions from the *OperatingSystem* *Circus* process through the channel *os_send_mHM_table* and then assigns these values to the state variables *mHM* and to *sysError*. This *Circus* process still does not cover how the health monitor behaves according the list of errors, *sysError*, and the recovery actions, *mHM*, for the module level.

$$\textbf{process}\ \ ModuleHM \mathrel{\widehat=} \textbf{begin}$$

$$\begin{array}{l} \textbf{state}\ \ ModuleHMSt == [modHM : \mathrm{seq}_1\, A653\_ModuleHMTableType; \\ \qquad\qquad sysError : \mathrm{seq}_1\, A653\_ErrorIdentifierType] \\ \bullet\ (os\_send\_modHM\_table?mp?se \longrightarrow modHM, sysError := mp, se) \\ \qquad\qquad \triangle moduleEnd \longrightarrow \textbf{Skip} \end{array}$$

end

We define below the *Circus* channel set *ModuleHMChannels* for communication between the *OperatingSystem* and the *MultiPartitionHM* *Circus* process.

$$\mathbf{channelset}\ ModuleHMChannels == \{\!|\ os\_send\_modHM\_table, moduleEnd\ |\!\}$$

Finally, we define the *OS_Layer*, according to the Fig. 3.1, with the parallelism between the *OperatingSystem* *Circus* process and the *ModuleHM* *Circus* process, communicating through the channels that composes the *ModuleHMChannel*.

$$\begin{array}{l}\mathbf{process}\ OS\_Layer \mathrel{\widehat=} module : Module \\ \quad \bullet \left(\begin{array}{l}\left(\begin{array}{l}OperatingSystem(module)\\ [\![\{\!|\ updateClock\ |\!\}]\!]\\ Timer\end{array}\right)\\ [\![ModuleHMChannels\ ]\!]\ ModuleHM\end{array}\right)\end{array}$$

We conclude the construction of our *Circus* model of the IMA architecture with the *IMA_Module* *Circus* process which represents the structure of the three top layers of the IMA architecture: operating system, the Application Executive (APEX), and the set of partitions. We model the *IMA_Module* *Circus* process as a parallel composition of the three *Circus* process presented in this chapter: *Partition_Layer*, *APEX_Layer*, and *OS_Layer*. The communication between *Partition_Layer* and *APEX_Layer* is made through channels of the *Circus* channel set *PartitionApex*. Similarly, the communication between *APEX_Layer* and *OS_Layer* is made through the channels that compose the *ApexOS* *Circus* channel set. In our formalisation, we distribute the content of the configuration tables through the components of the *IMA_Module* *Circus* processes. The model is based on the ARINC 653 document, which is equivalent to the structure illustrated in Figure 3.1.

$$\begin{array}{l}\mathbf{process}\ IMA\_Module \mathrel{\widehat=} module : Module \\ \bullet \left(\begin{array}{l}OS\_Layer(module)\\ [\![ApexOs]\!]\\ \left(\begin{array}{l}APEX\_Layer\left(\begin{array}{l}module.HealthMonitoring.MultiPartitionHM,\\ module.HealthMonitoring.SystemErrors\end{array}\right)\\ [\![PartitionApex]\!]\\ Partitions\_Layer\left(\begin{array}{l}1\mathinner{\ldotp\ldotp}\#\,module.Partitions,\\ module.HealthMonitoring.PartitionHM,\\ module.HealthMonitoring.SystemErrors\end{array}\right)\end{array}\right)\end{array}\right)\end{array}$$

In the previous sections of this chapter we presented how we model the IMA architecture using *Circus*. So far, we have presented the structure of the partitions and how they communicate with the lower levels of the IMA architecture. We also have presented the structure of the Application Executive. Finally, we model the operating system, including scheduling capabilities and how we capture the temporal partitioning properties of the architecture. In the next section, we present the steps taken in order to validate the *Circus* model by translating it into an equivalent CSP version in order to perform model checking and animation using existing tools.

3.5 Translating the *Circus* model into CSP

In this section we present an overview of the translation process from the *Circus* model of the IMA architecture into CSP in order to validate and animate the model using FDR and ProBE. During the translation, parts of the *Circus* specification had to be adapted in order to have an equivalent CSP specification.

In the translation from *Circus* to CSP, we need to define the types in our model as subsets of the types used in the *Circus* model. Types must be finite and small in order to avoid state explosion. As we are dealing with the model checker FDR, we need to avoid the generation of infinite states during the compilation process. For example, we define the type $DecOrHexValueType$ as a subset of natural numbers, ranging from 0 up to 10.

```
DecOrHexValueType = {0..10}
```

Another restriction related to the translation of our model into CSP is that we cannot define as an abstract type the type $String$ defined in our *Circus* specification. We define free types to construct a set of possible values for our types. For example, the free type $StringAccessRights$ is used in the definition of memory areas for a partition, and is illustrated below.

```
datatype StringAccessRights = READ_ONLY | READ_WRITE | NoAccessRights
```

Then we model the type $String$, illustrated below, as free types such as the presented above $StringAccessRights$.

```
datatype String = SNT.StringNameType | SAR.StringAccessRights
            | SMRT.StringMemoryRegionType | SMRN.StringMemoryRegionName
            | SPNT.StringSAMPLING_PORT_NAME_TYPE | QPNT.StringQUEUING_PORT_NAME_TYPE
```

The translation of the free type $ARINC\_CONSTANTS$ is pretty straightforward from our *Circus* model.

```
datatype ARINC_CONSTANTS = COLD_START | WARM_START | COLD_RESTART | WARM_RESTART
            | IDLE | NORMAL | ERROR_MODE | IGNORE | SHUTDOWN | RESET
            | ARINC_CONSTANTS_NULL
```

The translation of the subsets $OPERATING\_MODE\_TYPE$ and $OPERATING\_INIT\_MODE\_TYPE$ is made using the operation *diff* which results in the subtraction of elements in curly brackets, $\{COLD\_RESTART, WARM\_RESTART, ...\}$ from the set of $ARINC\_CONSTANTS$.

```
OPERATING_MODE_TYPE = diff(ARINC_CONSTANTS,
               {COLD_RESTART, WARM_RESTART, ERROR_MODE,
                   IGNORE, SHUTDOWN, RESET, ARINC_CONSTANTS_NULL })

OPERATING_INIT_MODE_TYPE = diff(ARINC_CONSTANTS,
               {COLD_RESTART, WARM_RESTART, IDLE, NORMAL, ERROR_MODE,
                   IGNORE, SHUTDOWN, RESET, ARINC_CONSTANTS_NULL})
```

3.5.1 Specification of the types for the Configuration Tables

We translate Z schemas into CSP as tuples. Each component of a Z schema is translated as an element of a tuple. For example, the schema *Partition* is translated as a tuple of four components where, for example, the first component of the tuple is of type *A653_PartitionBaseType* and the fourth component is of type *PartitionPort*.

```
nametype Partition = (A653_PartitionBaseType, A653_PartitionPeriodicityType,
                      MemoryRegion, PartitionPort)
```

In order to avoid state explosion, we also need to provide subsets of the non-empty injective sequences defined in our *Circus* model. First, we define instances of elements of those sequences. For example, we present below three instances of partitions, *Partition*1, *Partition*2, and *Partition*3 of type *Partition*.

```
Partitions1 = ((SPtN.PartitionName1, 1), (2, 1), PartitionPort1, MemoryRegionType1)
Partitions2 = ((SPtN.PartitionName2, 2), (1, 1), PartitionPort2, MemoryRegionType1)
Partitions3 = ((SPtN.PartitionName3, 3), (3, 1), PartitionPort3, MemoryRegionType1)
```

Instead of translating types that are non-empty sequences of defined types, we create a set with a few examples of possible sequences of that particular type. We illustrate this with the example of the type *PartitionType*, which is a set of three sequences of partitions.

```
nametype PartitionsType = {<Partitions1>, <Partitions1, Partitions2>,
                           <Partitions1, Partitions2, Partitions3>}
```

We translate the schema *Module* similarly to the definition of *Partition*. It is a tuple containing a name, a sequence of partitions, the sequence of execution of the partitions, and the information regarding the health monitor for all the levels in the architecture.

```
nametype Module = (NameType, PartitionsType, ScheduleType, HealthMonitoringType)
```

In *Circus*, when we want to have the values of a component of a schema, we simply describe it as the name of the variable, followed by a character '.'(dot) and the name of the component. For example, if we want to access the *Name* of a component *AModule* of type *Module*, we describe it as *AModule.Name*. However it is not possible to use that type of construct in CSP. We need to adopt a different approach for the translation: we define an auxiliary operation, for example, *getModuleName* illustrated below, in which the input is a tuple of type *Module* and the result is the name of the module. The equivalent result in CSP to *AModule.Name* is *getModuleName*(*AModule*).

```
getModuleName((Name, Partitions, Schedules, HealthMonitoring)) = Name
getPartitions((Name, Partitions, Schedules, HealthMonitoring)) = Partitions
getPartSchedules((Name, Partitions, Schedules, HealthMonitoring)) = Schedules
getHealthMonitoring((Name, Partitions, Schedules, HealthMonitoring)) = HealthMonitoring
```

3.5.2 Translation of the *A653_Partition* *Circus* process

In *Circus* the actions of a process is enclosed between a **begin** token and a '•', and the main action of the process is included after the '•'. We translate *Circus* actions of processes into CSP by defining the process as local definitions using the construct **let within**. Moreover, the main action of the process is defined after the **within** token. As an example, the structure of the *Circus* process *A653_Partition* is translated into CSP as illustrated below: *Circus* actions such as *GetTime* are included after the **let** token and then the main action is defined after **within**.

```
A653_Partition(partitionId, pHM, sysError) =
    let
    GetTime = proc_req_system_time.partitionId?processId ->
        part_req_system_time.partitionId ->
        part_get_system_time.partitionId?t ->
        proc_get_system_time.partitionId.processId!t ->
        return_code.partitionId?rc -> SKIP

    ...

    within moduleInit -> (ExecPartition /\ (moduleEnd -> SKIP))
```

3.5.3 Translation of *Circus* processes containing state

Differently from *Circus*, there is no direct translation of the state of a *Circus* process into CSP. We can represent the state of the process by creating a new process, for the state, in which every component of the *Circus* state is defined as input of the process in CSP and the access to those values is made through CSP channels, allowing the main action of the translated process to access the values of the state components and also assign new values to these components.

In general lines, whenever a value of the state is updated, a channel prefixed by '*set_*' receives the value from the main action and the state process recurses with the new value received. However, when the main action is going to access any of the components of the state, it made by request through a channel prefixed by '*get_*' and then the state process recurses without modifying any of the values of the components. Each '*get_*' or '*set_*' channel is offered in an external choice. After the communication of the channel, the state process recurses. Then, the state process is put in parallel with the main action of the translated process, communicating via the channels of the state process.

We illustrate this with the example of the process *PartitionHM*, translated from *Circus*. In the example, a CSP process *PartitionHMSt*, representing the state of the process *PartitionHM*, is modelled with the two components of the state as inputs of the process. The channels *set_partHM_table* and *get_partHM_table* will, respectively, update and offer the values of the state. Moreover, as the main process of the *PartitionHM* process can be interrupted by the channels *endPartition* and *moduleEnd*, these channels are also offered in the external choice of the state process *PartitionHMSt*, and their communication will result in the termination of the process by a *SKIP*.

The state process *PartitionHMSt* and the main action *MainPartitionHM* are put in parallel, synchronising on the channels of the channel set *PartitionHMStChannels*, presented below. Moreover, we define a initial set of values for the initialisation of the state process, and the channels used only for communication between the state and the main action is hidden from outside the *PartitionHM* process.

```
PartitionHMStChannels =
        {|set_partHM_table, get_partHM_table, moduleEnd, endPartition|}

  PartitionHM(partitionId) =
      let
      PartitionHMSt(pHM,sysError) =
          set_partHM_table?phm?se ->
              PartitionHMSt(phm, se)
          [] get_partHM_table!pHM!sysError ->
              PartitionHMSt(pHM, sysError)

      MainPartitionHM =
          (part_send_partHM_table.partitionId?phm?se ->
              set_partHM_table!phm!se ->  SKIP)

      within ((MainPartitionHM [|PartitionHMStChannels|]
                  PartitionHMSt(PartitionHMTableInit, SystemErrorInit))
                      /\ ((moduleEnd -> SKIP)
                          [] (endPartition.partitionId -> SKIP))
              ) \{|set_partHM_table,get_partHM_table|}
```

Before starting the validation of our *Circus* model of the IMA architecture, we had modelled the partition layer as an interleaving of the partitions. It was justified by the fact that the ARINC 653 standards specifies that partitions can not communicate with each other without requesting communication via the APEX services.

However, whilst animating the CSP model, translated from *Circus*, using ProBE, we noticed that: all partitions should be initialised simultaneously; after the initialisation phase, the partitions should be available all at the same time; and all partitions should be interrupted at the end of execution of the module simultaneously. The use of interleaving would cause nondeterminism during the execution of the system, since any partition could be ready to request the APEX services while others were still in the initialisation phase. We then concluded that all partitions should synchronise at the same time with the APEX and the operating system on the channels *moduleInit*, *moduleEndInit*, and *moduleEnd*.

```
  Partitions_Layer(partitionIds, partHM, sysError) =
      ([| {|moduleInit, moduleEnd, moduleEndInit|} |] partitionId: partitionIds
          @ ((A653_Partition(partitionId, getPartHM(partHM, partitionId), sysError)
              [|PartitionHMChannels|] PartitionHM(partitionId)
                  ) \ {| part_send_partHM_table |}))
```

3.5.4 The Operating System process in CSP

The most complex part of the translation of the *Circus* model into CSP consists in the translation of the *Circus* process *OperatingSystem*. We had to change the name of the input of the *OperatingSystem* CSP process since the name *module* is already being used for the definition of the type *Module* and the parser used in FDR is not case sensitive. We then define the variable *modl* as the input of the *OperatingSystem* CSP process. We start the translation by creating the equivalent process to the *Circus* state, *OSSt*, of the *OperatingSystem* process. During the translation of the state *OSSt* into CSP, we define that the access and update of the values of the state components is made via new channels, introduced in our CSP model for the communication between the state process and the rest of the translated CSP process of the *OperatingSystem*.

At the beginning of the execution of the state process *OSSt*, we include a restriction, the invariant of the *OSSt* process, equivalent to the *Circus* model. At each recursion of the process, the invariant is checked: the state process will terminate if the invariant is broken. Moreover, for each component of the state, we define a '*get_*' and '*set_*' channel, offered in an external choice, in order to allow the *OperatingSystem* main action to have access and update the values of the state components.

```
OperatingSystem(modl) =
    let
    OSSt(PartitionsVariables, major_time_frame, system_time, current_partition,
        current_partition_id, sampling_ports, queuing_ports) =
            if (# PartitionsVariables == # getPartitions(modl))
            then ((set_PartitionsVariables?pv ->
                    OSSt(pv, major_time_frame, system_time,
                        current_partition, current_partition_id,
                        sampling_ports, queuing_ports))
                [] (get_PartitionsVariables!PartitionsVariables ->
                    OSSt(PartitionsVariables, major_time_frame, system_time,
                        current_partition, current_partition_id,
                        sampling_ports, queuing_ports))
                [] ...)
            else SKIP
    ...
```

We present one of the auxiliary operations for the translation of the *Circus* model of the operating system into CSP. The operation *OrderSeq* is used by the CSP process *MajorTimeFrame* in order to return the partition, within the sequence of partitions of a module, with the highest *Period*. If the sequence of partitions has a single partition, the *OrderSeq* will return that partition.

```
OrderSeq(<e>)=e
OrderSeq(<e>^x)=
    let OrderSeq1(<e>^s,m) =
            if (getPeriod(getPartitionPeriodicity(e))
                    > getPeriod(getPartitionPeriodicity(m)))
            then OrderSeq1(s,e) else OrderSeq1(s,m)
        OrderSeq1(<>,m) = m
    within OrderSeq1(x,e)
```

As we cannot have direct access to the state components of the translated CSP model from the *Circus* processes, we use channels for communication with the state in CSP. For example, the *MajorTimeFrame* *Circus* action is translated into CSP using operations like *OrderSeq* in order to obtain the longest *Period* of a partition. However, as the *major_time_frame* state component is updated with the result of that operations, we update that component in the state process *OSSt* by communicating the new value of *major_time_frame* with the state via the channel *set_major_time_frame*.

```
MajorTimeFrame =
    set_major_time_frame!getPeriod(getPartitionPeriodicity(
                                OrderSeq(getPartitions(modl)))) -> SKIP
```

Another example of our translation strategy from *Circus* into CSP is the process *UpdateSystemTime*, which receives the value x from the *Timer* process via the channel *updateClock* and then sends x to the state process via the channel *set_system_time*. We see here that instead of using an assignment to a state variable, for example *system_time* := x, the value is sent to the state process and updated after the synchronisation.

```
UpdateSystemTime = updateClock?x -> set_system_time!x -> SKIP
```

In our translation model, there are several channels that uses state values in its communications. It is the case of the state component *current_partition_id* of the *OperatingSystem* *Circus* process. It is used to indicate the identifier of the partition that can have access to the *APEX* services. We can have access to such value in our CSP model by including it in the scope of the translated *Circus* action by first receiving that value from the state process via an input channel, for example *get_current_partition_id* and then we enclose the action of the process within parenthesis. Thus, every communication or internal calculation of the process can use the value of the state component, like *current_partition_id*.

The *Circus* action *OSGetTime* is translated into CSP as follows: the channel *get_current_partition_id* receives from the state process the identifier of the current partition in execution, then it receives from the *APEX* a signal that the partition with identifier *cid* is requesting the time of the system. Next, the process *UpdateSystemTime* process is executed. Finally, the current value of the time is received from the state process via *get_system_time* and then it is sent back to the *APEX* through the channel *apex_get_system_time*.

```
OSGetTime = get_current_partition_id?current_partition_id ->
    (apex_req_system_time.current_partition_id -> UpdateSystemTime;
        get_system_time?st -> apex_get_system_time.current_partition_id!st -> SKIP)
```

The translation of the action *OSGetPartitionStatus* does not use the auxiliary *Circus* action *PartitionStatus*. In CSP, we start the equivalent process by requesting that the state process provide the identifier of the current partition in execution. This communication is used for every translated *Circus* action into CSP. Then, the *APEX* requests the current status of the partition with identifier *current_partition_id*. Following, the sequence of *PartitionsVariables*, *pv*, is received from the state process. Finally, if the number of elements of the sequence *pv* is greater than zero, the channel *apex_get_partition_status* will send to the *APEX* the tuple of type *PartitionsVariables* of the partition *current_partition_id* with the values of that partition. We are able to construct the tuple of type *PartitionsVariables* by using the auxiliary operations such as *getPid* and *getPeriod*, which returns these values from the input tuple.

```
OSGetPartitionStatus =
    get_current_partition_id?current_partition_id ->
        (apex_req_partition_status.current_partition_id ->
        get_PartitionsVariables?pv ->
        if (#pv > 0)
        then apex_get_partition_status.current_partition_id!(
                getPId(getPartitionBase(
                            getPartition(getPartitions(modl), current_partition_id))),
                getPeriod(getPartitionPeriodicity(
                            getPartition(getPartitions(modl), current_partition_id))),
                getDuration(getPartitionPeriodicity(
                            getPartition(getPartitions(modl), current_partition_id))),
                (getPVOperMode(getPS(pv, current_partition_id)),
                  getPVStartCond(getPS(pv, current_partition_id)),
                  getPVLockLvl(getPS(pv, current_partition_id)))) -> SKIP
        else SKIP)
```

In our *Circus* model, we use *Circus Time* constructs in order to capture the scheduling capabilities of the IMA architecture. For example, we use the **Wait** construct in order to specify the time delay before the execution of the next partition according to the schedule defined in the configuration tables. We also use the *Circus Time* timeout construct in order to interrupt the execution of the actual partition when its allocated time slice has expired.

In order to capture the timed behaviour of our *Circus* specification in CSP, which natively does not allows us to specify timed models, we need to explicitly incorporate the passage of time in our specification. One solution is to include an event *tock*, which marks the passage of time. That approach is called *tock-CSP* [51]. Every process that relies on timing constraints must synchronise with the *tock* event. With *tock*, we are able to define equivalent processes in CSP for the *Circus Time* constructs **Wait** and timeout [51].

We present first the process **Wait**, which is modelled as follows: the input n defines how many tock events will occur before the end of the execution of the **Wait** process. In other words, the process **Wait** will delay n time units the execution of the next process in the system.

```
Wait(n) = if n>0 then tock -> Wait(n-1) else SKIP
```

We model the timeout construct in CSP in the following way. The process *Timeout* has three inputs: process A, process B, and t time units for the timeout. We first define a local process that recurses decrementing t time units, represented by occurrences of *tock* event. When the counter reaches zero, an event *timeout* occurs and the *Counter* process terminates. The main action of the *Timeout* process is the parallel composition between the process $A \vartriangle \mathit{timeout} \longrightarrow B$ and the *Counter* process, synchronising on timeout and tock events. The process $A \vartriangle \mathit{timeout} \longrightarrow B$ means that A is executed until a *timeout* signal from the process *Counter*, interrupting A and starting the execution of B. An example of using timeout is presented in [51], however, a definition of construct *Timeout* using *tock-CSP* is not presented. Our definition of a *Timeout* differs from [57], in which, an external choice is used and A can only be executed if it engages a communication or if it terminates before the *Wait* period d. In our example, A is executed until an interruption (/\) with a signal *timeout*, triggered by $Counter(t)$ followed by the process B after t elapses.

```
Timeout(A,B,t) =
    let Counter(t) =
        if (t>0) then tock -> Counter(t-1)
        else timeout -> SKIP
    within ((A /\ timeout -> B) [|{|timeout,tock|}|] Counter(t)) \ {|timeout|}
```

Having modelled the equivalent processes to the *Circus Time* constructs, we can now translate the *ExecOS* process of the *OperatingSystem*, which is the core process of the operating system.

The execution of the *ExecOS* process starts by updating the system time component of the state process. Next, the process *NextPartition* calculates which of the partitions is to be executed next. Then it recurses as follows: at each iteration, if the offset of the next partition to be executed is greater than the current system time, then the *Wait* process is executed during the interval between the current time until reaching the offset of the partition and then it recurses. However, if the offset of the partition to be executed in the sequel is equal to the current system time, then the *Timeout* process is executed with the following inputs: the process *ExecOSServices*; *InterruptActions*, which is simply a signal *interruptPartition* to the partition of identifier *current_partition_id*; and the duration of execution of that partition.

```
ExecOS =
    let
    X = (UpdateSystemTime ; NextPartition ;
        get_system_time?system_time ->
        (get_current_partition_id?current_partition_id ->
        (get_major_time_frame?major_time_frame ->
        (if major_time_frame > 0
         then get_current_partition?current_partition ->
            (if (getPartitionTimeWindowOffset(current_partition) >
                            (system_time % major_time_frame))
             then Wait(getPartitionTimeWindowOffset(current_partition) -
                            (system_time % major_time_frame)) ; X
             else if (getPartitionTimeWindowOffset(current_partition) ==
                               (system_time % major_time_frame))
                  then Timeout(ExecOSServices,
                          InterruptAction(current_partition_id),
                          getPartitionTimeWindowDuration(current_partition)) ; X
                  else SKIP)
         else SKIP ))))
    within MajorTimeFrame; X
```

As we translate the state of the *OperatingSystem* *Circus* process as a state process in CSP, we need to define the main action of the *Circus* process as a particular CSP process, in order to put it in parallel with the state process *OSSt*.

We define here the main action of the *OperatingSystem* process as the CSP process *OSMainAction* which is similar to the main action of the *Circus* process. The difference is how we access the values of the health monitor from the input variable *modl*, using here, auxiliary operators such as *getModuleHMTable* and *getSystemError*.

```
OSMainAction =
    moduleInit ->
    os_send_modHM_table!(getModuleHMTable(getHealthMonitoring(modl)))!(
                    getSystemError(getHealthMonitoring(modl))) ->
    (ExecOS /\ moduleEnd -> SKIP)
```

We conclude the translation of the *OperatingSystem* *Circus* process into CSP with the parallelism between the main action of the *OperatingSystem*, represented by the process *OSMainAction*, and the state process *OSSt*, with the initial values of the state defined as inputs of the process. These values are not specified in the ARINC 653 document, but are simply defined by us, and does not affect the execution of the process. All the state values are updated in the during of execution of the *OSMainAction*. The two processes synchronise on the channel set *OSStchannels*, which comprises the channels for obtaining and updating the values of the state components. However, the communication between the state process *OSSt* and the *OSMainAction* is hidden from outside the *OperatingSystem* process.

```
within (OSMainAction [|OSStchannels|]
            OSSt(<PartitionVariables1>, 0, 0, PartitionTimeWindowNull,
                0, {(0,SPNT.SPNone)}, {(0,QPNT.QPNone)})) \ OSStchannels
```

The CSP process equivalent to the *Circus* process *OS_Layer* is presented below: the *OperatingSystem* process is put in parallel with the *ModuleHM* process, translated similarly to the *PartitionHM* process previously presented here, and the *Timer* process, which the translated process uses the defined *Wait* and *Timeout* processes for timing management.

```
OS_Layer =
    ((ModuleHM [|ModuleHMChannels|]
            (OperatingSystem(amod)\OSStchannels)) \ {|os_send_modHM_table|})
                [|{|updateClock,moduleInit,moduleEnd|}|]
                    (Timer \ {|get_clock,set_clock|})
```

We conclude the translation of our *Circus* model of the IMA architecture with the equivalent CSP process of the *Circus* process *Module*. It consists of the parallel composition between the *OS_Layer*, the *Partitions_Layer* and the *APEX_Layer*, synchronising on the channels belonging to the channel sets *ApexOS* and *PartitionApex*.

```
Module(modl) =
    OS_Layer [| ApexOs |]
        (Partitions_Layer({1..#getPartitions(modl)},
            getPartitionHMTable(getHealthMonitoring(modl)),
            getSystemError(getHealthMonitoring(modl)))
        [| PartitionApex |]
            APEX_Layer(getSystemError(getHealthMonitoring(modl)),
                getMultiPartitionHMTable(getHealthMonitoring(modl))))
```

In this section we presented the translation of our model of the IMA architecture from *Circus* to CSP[2]. In the next section, we discuss the validation of the model using FDR.

3.6 Validation of the model using FDR

We validated our CSP model of the IMA architecture using FDR, with assertions regarding deadlock freedom, livelock freedom, determinism and termination. These assertions were checked for each translated process individually, for example, for the *A653_Partition* and the *PartitionHM* processes. Then, the checks were performed for the parallelism between the *Circus* processes, such as the *Partition_Layer* and *OS_Layer*.

With help of these assertions, we were able to define the channel sets used for the communication between the layers of the architecture, specified as processes. In particular, the assertions regarding the termination of the processes has shown the expected result: none of the partitions processes or the *APEX* process should terminate by itself. We see in FDR that these processes never terminate. However, when the assertion for termination

[2]The full version of the CSP specification can be found elsewhere at http://www.cs.york.ac.uk/∼artur/ima-spec.csp

is assessed for the *Module*(*modl*) process, we see that the system terminates, which means that the operating system is the one that triggers the termination of the other processes.

Moreover, as mentioned before in this section, by using the animator ProBE, we could identify that instead of interleaving the partitions within a module, they should be in parallel, synchronising during the initialisation of the module, at the end of the initialisation phase, and at the point in which the module is switched off.

Verifying the assertions about the specification in FDR required a large amount of time due to the complexity of the system. Since we are dealing with a large number of processes in parallel and their communication channels, some of the assertions required more than 15 hours to verify. The amount of time was increased with the inclusion of *tock* events and timing constructs in the specification.

There is one thing we need to have in mind when performing model checking of *Circus* specifications: we need to avoid state explosion. In FDR, the translated specification of the IMA architecture into CSP can cause state explosion due to the infinite number of possible states of the system. We then have to restrict, in the CSP specification, the types to a small subset of the original types specified in *Circus*.

As a consequence, we can not explore a wide number of examples of configuration tables for IMA modules. For instance, the inclusion of more than two partitions in the example of configuration table used in this dissertation increases exponentially the number of internal calculations in FDR, and consequently, increases the amount of time required by FDR to verify the assertions.

Considering that this specification includes only four of the dozens of APEX services such as those for partition and process management, interpartition and interpartition communication and health monitoring, and we use in our assertions a small example of the configuration tables, it looks like that the amount of time required to validate this model including new APEX services will be highly increased. One solution for this problem might be the use of compression techniques [48] in FDR, which might help to reduce the amount of time verify for the internal calculations of the model checker.

Until now thre is no automatic translation tool from *Circus* into CSP, aiming at model checking with FDR. And because we are translating the specification by hand, we should pay attention to several details of the equivalences between the two formalisms. In most of the *Circus* constructs, we have a pretty straightforward translation into CSP, as we know that *Circus* is derived from CSP.

We have to closely look at the inclusion of the notion of a state in the translated CSP processes from *Circus*. We include extra CSP channels for the communication between the CSP internal process, that describes the behaviour of the system, and the state of the process.

Moreover, we have also to carefully analize and design equivalent constructs for the predicates of Z schema expressions, which are not always very straightforward, requiring sometimes the creation of auxiliary functions in order to establish the equivalence between the model in CSP and the one in *Circus*.

In summary, we can cite a few basic requirements in order to design a tool for automatic translation from *Circus* into CSP: (1) Identify the sets of channels used in the *Circus* specification for communication between the *Circus* processes. (2) Identify the state of the *Circus* process, its components and invariants, then create a new set of channels for the communication between the state and the components of the *Circus* process, offering and updating the values of the components of the state, through internal communication using CSP channels hidden from outside the equivalent CSP process from the *Circus* process. (3) In case of the use of Z schema expressions, identify the predicates of the operation and design equivalent constructs of them in CSP.

In our case, as we use *Circus Time* constructs in our specification, we can not have direct translation into CSP as FDR does not support timed CSP. However, we have to manually deal with the translation of *Circus Time* constructs into CSP by using *tock*-CSP.

Specification of specific properties during the execution of the model such as examples of expected and unexpected sequences of execution of IMA partitions can be described as part of future work.

3.7 Final Considerations

We have presented in this chapter how we use *Circus* in order to specify the three top layers of the IMA architecture and how we validate it in FDR using a translated version of the model into CSP. During the translation from *Circus* to CSP, we can see the similarities between the syntax of the two languages, since we know that *Circus* is a combination of Z and CSP. We translate choice of actions, parallelism and interleaving between processes keeping the same order of the execution flow of the model just as described in *Circus*.

Types defined in *Circus* as natural numbers are redefined in CSP as small finite subsets in order to avoid state explosion whilst using FDR. Abstract types are translated into CSP as free types with a few constants used for modelling the configuration tables. Free types specified in *Circus* are translated into CSP as datatypes. Moreover, schema types are translated into nametypes in which the components of the schema are structured in a tuple. We also define functions in CSP in order to obtain the values of these nametypes.

We also keep the same name for the CSP channels from the *Circus* specification. The notation used for describing the types of channels that communicate more than one value of the same type in CSP is slightly different from *Circus*: the character '$\times$' (cross) is replaced by a '.' (dot).

Circus processes are translated as CSP processes. The actions of the *Circus* process are described as local processes and are located between the construct **let within**. Moreover, the main action of the *Circus* process is described after the **within**.

Capturing the state of *Circus* processes requires the creation of another local process. We create a state process of the CSP process in which the components of the *Circus* state are translated as inputs of the state process. Moreover, the access and update of values of the state is made through communication using new channels for getting and setting the values of the state.

Parallelism, interleaving and choice are captured in CSP in a similar syntax from the *Circus* notation. The translation of the *Circus Time* model is made with help of *tock-CSP*.

We use equivalent CSP notation for the *Wait* and *timeout* *Circus Time* constructs [51].

In the next section we present the conclusions regarding the current state of the above presented work and directions for future work.

Chapter 4

Conclusions

In this dissertation we present a formal model of the IMA architecture. In particular we use *Circus* as a formal language to model and validate the IMA architecture. The resulting model captures the temporal partitioning feature of the IMA partitions within a module.

4.1 Summary and Contributions

In this dissertation we provided an overview of the IMA architecture, detailing its components and features. We also present an overview of existing approaches on certification and verification of IMA systems. Moreover, we present a brief survey of formal languages for the design of concurrent systems. By using *Circus*, we are able to formally specify the IMA architecture including scheduling capabilities used for temporal partitioning using *Circus Time* constructs.

Afterwards, we provided a *Circus* model [24] of three layers of components of the IMA architecture. It comprises a model of the operating system layer, the application executive (APEX) layer and the partitions layer. The complete *Circus* model of the IMA architecture is included in the Appendix A.

We first present the partitions layer, which represents the set of partitions that are executed within an ARINC 653 module. They are modelled in *Circus* as a generalised parallel composition between each $A653\_Partition$ and its health monitor, the $PartitionHM$ *Circus* process, defined as a new *Circus* process called $Partitions\_Layer$.

The APEX layer is modelled in *Circus* as the parallel composition of two *Circus* processes: the $APEX$, which manages the requests of its services from the partitions; and the $MultiPartitionHM$, which behaves according to its recovery actions, in case of detected errors.

And finally, the operating system layer is modelled as the parallel composition of three *Circus* processes. The $OperatingSystem$, contains records of the partitions requirements, and behaves as described in the ARINC 653 documents, where the partitions are executed within a schedule. We modelled the scheduling capability of the operating system using *Circus Time* constructs, using the timed interrupt construct to interrupt the execution of partitions. Moreover the *Circus* process $Timer$ provides to the $OperatingSystem$ the actual time of the system. Finally, the $ModuleHM$ receives the information regarding health monitoring for the module level: its behaviour regarding errors and failures is part of our future work.

We were able to validate our model using FDR with assertions regarding deadlock freedom, livelock, nondeterminism and termination check which is not mentioned in none of the related work in verifying avionics systems, presented in Section 2.2.2.

4.2 Future Work

The next steps for this work consist in expanding the presented *Circus* model of the architecture. We need to model the ARINC processes within a partition, and model a scheduler for these process within a partition, based on the ARINC 653 document. Moreover, we need to extend the range of APEX services, including the services for process management, time management, interpartition communication, intrapartition communication, and health monitoring. We also need to model the behaviour of the health monitor *Circus* processes for the module, the APEX and the partitions.

As we are dealing with a complex specification, we can see that it becomes more difficult to validate it in FDR after including new services of the APEX and using larger examples of configuration tables. We know that compression techniques may be helpful in order to reduce the time required from FDR in order to verify the assertions about the specification. Although, we are not sure about how much time it would be reduced by using these techniques in the CSP specification of the IMA architecture presented here.

In this work, we address the specification of time requirements of IMA partitions. However, the specification of spatial partitioning aspects of IMA partitions is also an interesting subject of research.

An interesting piece of future work is that, once having the *Circus* model of the IMA architecture extended to capture the internal computations of the partitions, i.e., including scheduling and communication services for the processes as well as including the health monitor recovery services, it can be used as a *Circus* library in order to be able to verify implementations of IMA programs.

By taking advantage of existing work related to *Circus*, it might be possible to prove the refinement between *Circus* models of Simulink diagrams [36] and *Circus* models of concrete implementations of these programs [46] within an IMA module. The idea is to produce a model generation strategy which puts together the *Circus* model of the IMA architecture, presented in this dissertation with *Circus* models of implementations of IMA applications within an IMA module. The components of the IMA architecture presented here are fixed, and since their *Circus* model is always the same, they are verified only once. However, only specific components of the *Circus* model regarding IMA applications will need to be verified when included in the validated model of the architecture.

By using *Circus*, we can profit from its support for concurrent systems, which is one of the characteristics of IMA programs. Another benefit from adopting this approach as part of our work is that we can verify discrete time programs specified in Simulink. Moreover, we can benefit from modularity, which is essential for our approach, as we plan to verify the IMA applications in isolation from the architecture model. However, we need to extend this to *Circus Time*, as it is used during the construction of the *Circus* model of the IMA architecture, based on the ARINC 653 standards.

Another contribution for future work could be the extension of existing work [7] on the translation strategy of Simulink diagrams into *Circus* models in order to produce models with *Circus Time* constructs. By using *Circus Time* constructs we are able to capture the scheduling capabilities specified by ARINC 653, producing *Circus* models of IMA systems using *Circus Time* constructs such as **Wait** and timeouts. Moreover, it might be necessary to analyse the existing refinement strategy [8] for *Circus* to the context of avionics systems: it would be necessary to develop new refinement laws regarding the Simulink libraries for this subject.

Finally, an interesting piece of future work outside the scope of this thesis would be the use of an industrial-scale case study that could help us to evaluate the scalability of our approach. Moreover, it would allow us to see how automated is our approach and how we could develop a tool that automates our proposed model generation strategy of implementations of IMA programs.

Appendix A

A *Circus* Model of the ARINC 653 Components

A.1 Types

$$iseq_1[X] == \{ s : \mathrm{seq}\, X \mid s \neq \langle\rangle \land s \in \mathbb{N} \rightarrowtail X \}$$

A.1.1 Basic types definition

1. Definition for numbers

   $$DecOrHexValueType == \mathbb{Z}$$

2. Definition for Strings

   $$[String]$$
   $$NameType == String$$

3. Definition of Boolean

   $$Boolean ::= TRUE \mid FALSE$$

4. Definition of Time

   $$SYSTEM\_TIME\_TYPE == DecOrHexValueType$$

5. Definition of Partitions and Process Identifiers

   $$PARTITION\_ID\_TYPE == DecOrHexValueType$$
   $$PROCESS\_ID\_TYPE == DecOrHexValueType$$

6. Definition of messages provided in the end of the execution of an APEX service

$$\begin{aligned} RETURN\_CODE\_TYPE ::= {} & NO\_ERROR \mid NO\_ACTION \mid NOT\_AVAILABLE \\ & \mid INVALID\_PARAM \mid INVALID\_CONFIG \\ & \mid INVALID\_MODE \mid TIMED\_OUT \end{aligned}$$

7. Definition of ARINC constants, including operating modes and recovery actions.

$$\begin{aligned} ARINC\_CONSTANTS ::= {} & COLD\_START \mid WARM\_START \mid COLD\_RESTART \\ & \mid WARM\_RESTART \mid IDLE \mid NORMAL \\ & \mid ERROR\_MODE \mid IGNORE \mid SHUTDOWN \\ & \mid RESET \mid ARINC\_CONSTANTS\_NULL \end{aligned}$$

8. Definition of Operating modes, subsets of $ARINC\_CONSTANTS$.

$$\begin{aligned} & OPERATING\_MODE\_TYPE == \\ & \quad ARINC\_CONSTANTS \setminus \{COLD\_RESTART, WARM\_RESTART, \\ & \qquad ERROR\_MODE, IGNORE, SHUTDOWN, \\ & \qquad RESET, ARINC\_CONSTANTS\_NULL\} \\ & OPERATING\_INIT\_MODE\_TYPE == \\ & \quad ARINC\_CONSTANTS \setminus \{IDLE, NORMAL, \\ & \qquad COLD\_RESTART, WARM\_RESTART, \\ & \qquad ERROR\_MODE, IGNORE, SHUTDOWN, \\ & \qquad RESET, ARINC\_CONSTANTS\_NULL\} \end{aligned}$$

9. Start Condition

$$\begin{aligned} START\_CONDITION\_TYPE ::= {} & NORMAL\_START \mid PARTITION\_RESTART \\ & \mid HM\_NORMAL\_START \mid HM\_PARTITION\_RESTART \end{aligned}$$

10. Lock Level

$$LOCK\_LEVEL\_TYPE == DecOrHexValueType$$

11. Errors and levels of error definition

$$\begin{aligned}
&\mathit{ErrorCodeType} ::= \mathit{DEADLINE\_MISSED} \mid \mathit{APPLICATION\_ERROR}\\
&\qquad \mid \mathit{NUMERIC\_ERROR} \mid \mathit{ILLEGAL\_REQUEST}\\
&\qquad \mid \mathit{STACK\_OVERFLOW} \mid \mathit{MEMORY\_VIOLATION}\\
&\qquad \mid \mathit{HARDWARE\_FAULT} \mid \mathit{POWER\_FAIL}\\
&\mathit{MODULE\_LEVELS} ::= \mathit{PARTITION} \mid \mathit{PROCESS}\\
&\qquad \mid \mathit{MODULE} \mid \mathit{MODULE\_LEVELS\_NULL}\\
&\mathit{PartitionHMTableErrorLevelType} ==\\
&\qquad \mathit{MODULE\_LEVELS} \setminus \{\mathit{MODULE}\}\\
&\mathit{MultiPartitionHMTableErrorLevelType} ==\\
&\qquad \mathit{MODULE\_LEVELS} \setminus \{\mathit{PARTITION}\}
\end{aligned}$$

12. Recovery Actions definition for Module and Partition levels

$$\begin{aligned}
&\mathit{ModuleLevelErrorRecoveryActionType} ==\\
&\qquad \mathit{ARINC\_CONSTANTS} \setminus \{\mathit{COLD\_RESTART}, \mathit{WARM\_RESTART},\\
&\qquad\qquad \mathit{COLD\_START}, \mathit{WARM\_START},\\
&\qquad\qquad \mathit{IDLE}, \mathit{NORMAL}, \mathit{ERROR\_MODE},\\
&\qquad\qquad \mathit{ARINC\_CONSTANTS\_NULL}\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{PartitionLevelErrorRecoveryActionType} ==\\
&\qquad \mathit{ARINC\_CONSTANTS} \setminus \{\mathit{COLD\_START}, \mathit{WARM\_START}, \mathit{NORMAL},\\
&\qquad\qquad \mathit{ERROR\_MODE}, \mathit{SHUTDOWN},\\
&\qquad\qquad \mathit{RESET}, \mathit{ARINC\_CONSTANTS\_NULL}\}
\end{aligned}$$

13. Port message types definition

$$\begin{aligned}
&[\mathit{MESSAGE\_ADDR\_TYPE}]\\
&\mathit{SAMPLING\_PORT\_NAME\_TYPE} == \mathit{NameType}\\
&\mathit{MESSAGE\_SIZE\_TYPE} == \mathit{DecOrHexValueType}\\
&\mathit{SAMPLING\_PORT\_ID\_TYPE} == \mathit{DecOrHexValueType}\\
&\mathit{VALIDITY\_TYPE} ::= \mathit{INVALID} \mid \mathit{VALID}\\
&\mathit{QUEUING\_PORT\_ID\_TYPE} == \mathit{DecOrHexValueType}\\
&[\mathit{QUEUING\_PORT\_NAME\_TYPE}]\\
&\mathit{QUEUING\_DISCIPLINE\_TYPE} ::= \mathit{FIFO} \mid \mathit{PRIORITY}\\
&\mathit{MESSAGE\_RANGE\_TYPE} == \mathit{DecOrHexValueType}\\
&\mathit{WAITING\_RANGE\_TYPE} == \mathit{DecOrHexValueType}\\
&\mathit{PortDirectionType} ::= \mathit{SOURCE} \mid \mathit{DESTINATION}
\end{aligned}$$

A.1.2 Configuration Tables Data Structure

1. Partitions

   (a) Partition Status Type Definition

$A653\_PartitionBaseType$

$Identifier : PARTITION\_ID\_TYPE$
$Name : NameType$

(b) Partition Memory Region Type Definition

$A653\_MemoryRegionType$

$Name, Type, AccessRights : NameType$
$Size : DecOrHexValueType$

$$MemoryRegion == iseq_1[A653\_MemoryRegionType]$$

(c) Partition Periodicity Type Definition

$A653\_PartitionPeriodicityType$

$Period : SYSTEM\_TIME\_TYPE$
$Duration : SYSTEM\_TIME\_TYPE$

(d) Partition Ports Types Definition

i. Port Base (used in both port types)

$PortBaseType$

$Name : NameType$
$MaxMessageSize : DecOrHexValueType$
$Direction : PortDirectionType$

ii. Sampling Port

$A653\_SamplingPortType$

$PortBaseType$

iii. Queuing Port

$A653\_QueueingPortType$

$PortBaseType$
$MaxNbMessage : DecOrHexValueType$

iv. Partition Ports Definition

$$\begin{aligned} \mathit{PartitionPort} ::= {} & \mathit{SamplingPort}\langle\!\langle \mathit{A653\_SamplingPortType}\rangle\!\rangle \\ & \mid \mathit{QueueingPort}\langle\!\langle \mathit{A653\_QueueingPortType}\rangle\!\rangle \end{aligned}$$

$$\mathit{PartitionPorts} == \mathit{iseq}_1[\mathit{PartitionPort}]$$

(e) Partition definition

$$\begin{array}{|l} \textit{Partition} \\ \hline \mathit{PartitionDefinition} : \mathit{A653\_PartitionBaseType} \\ \mathit{PartitionPeriodicity} : \mathit{A653\_PartitionPeriodicityType} \\ \mathit{MemoryRegions} : \mathit{iseq}_1[\mathit{A653\_MemoryRegionType}] \\ \mathit{PartitionPorts} : \mathit{iseq}_1[\mathit{PartitionPort}] \\ \hline \end{array}$$

(f) Sequence of Partitions definition

$$\mathit{PartitionsType} == \mathit{iseq}_1[\mathit{Partition}]$$

2. Partition Schedule

$$\begin{array}{|l} \textit{A653\_PartitionTimeWindowType} \\ \hline \mathit{PartitionNameRef} : \mathit{NameType}; \\ \mathit{Duration}, \mathit{Offset} : \mathit{DecOrHexValueType}; \\ \mathit{PeriodicProcessingStart} : \mathit{Boolean} \\ \hline \end{array}$$

$$\mathit{ScheduleType} == \mathit{iseq}_1[\mathit{A653\_PartitionTimeWindowType}]$$

3. Health Monitor

(a) Error Identifier Types

$$\begin{array}{|l} \textit{A653\_ErrorIdentifierType} \\ \hline \mathit{ErrorIdentifier} : \mathit{IdentifierValueType} \\ \mathit{Description} : \mathit{NameType} \\ \hline \end{array}$$

(b) Partition Health Monitor Definition

PErrorActionType

$ErrorIdentifierRef : IdentifierValueType$
$ErrorLevel : PartitionHMTableErrorLevelType$
$PartitionRecoveryAction : ModuleLevelErrorRecoveryActionType$
$ErrorCode : ErrorCodeType$

A653_PartitionHMTableType

$TableName : NameType$
$MultiPartitionHMTableNameRef : NameType$
$ErrorAction : \mathrm{seq}_1\ PErrorActionType$

(c) Multi-Partition Health Monitor Definition

MPErrorActionType

$ErrorIdentifierRef : IdentifierValueType$
$ErrorLevel : PartitionHMTableErrorLevelType$

A653_MultiPartitionHMTableType

$TableName : NameType$
$ErrorAction : \mathrm{seq}_1\ MPErrorActionType$

(d) Module Health monitor definition

MErrorActionType

$ErrorIdentifierRef : IdentifierValueType$
$ModuleRecoveryAction : ModuleLevelErrorRecoveryActionType$

A653_ModuleHMTableType

$StateIdentifier : IdentifierValueType$
$Description : NameType$
$ErrorAction : \mathrm{seq}_1\ MErrorActionType$

(e) Health monitor Definition

$HealthMonitoringType$

$SystemErrors : iseq_1[A653\_ErrorIdentifierType]$
$ModuleHM : iseq_1[A653\_ModuleHMTableType]$
$MultiPartitionHM : iseq_1[A653\_MultiPartitionHMTableType]$
$PartitionHM : iseq_1[A653\_PartitionHMTableType]$

4. Module Definition

$Module$

$Name : NameType$
$Partitions : PartitionsType$
$Schedules : ScheduleType$
$HealthMonitoring : HealthMonitoringType$

5. Partition Status Type Definition

$PartitionVariables$

$OPERATING\_MODE : OPERATING\_MODE\_TYPE$
$START\_CONDITION : START\_CONDITION\_TYPE$
$LOCK\_LEVEL : LOCK\_LEVEL\_TYPE$

$PARTITION\_STATUS\_TYPE$

$Identifier : PARTITION\_ID\_TYPE$
$Period : SYSTEM\_TIME\_TYPE$
$Duration : SYSTEM\_TIME\_TYPE$
$PartitionVariables$

A.2 *Circus* Channels

1. Channels for communication between the Operating System, APEX and the Partitions indicating the life stage of the ARINC Module.

$\mathbf{channel}\ moduleInit, moduleEndInit, moduleEnd$

2. Channels used by the Operating System *Circus* process to control the partitions execution.

$\mathbf{channel}\ initPartition, endPartition, reinitPartition : PARTITION\_ID\_TYPE$
$\mathbf{channel}\ execPartition, interruptPartition : PARTITION\_ID\_TYPE$
$\mathbf{channel}\ return\_code : PARTITION\_ID\_TYPE \times RETURN\_CODE\_TYPE$

3. Channels for communication between the APEX and the Operating System

$$
\begin{array}{l}
\textbf{channel}\ apex\_req\_sampling\_port\_id : PARTITION\_ID\_TYPE \\
\qquad \times SAMPLING\_PORT\_NAME\_TYPE \\
\textbf{channel}\ apex\_get\_sampling\_port\_id : PARTITION\_ID\_TYPE \\
\qquad \times SAMPLING\_PORT\_ID\_TYPE \\
\textbf{channel}\ apex\_req\_new\_partition\_mode : PARTITION\_ID\_TYPE \\
\qquad \times OPERATING\_MODE\_TYPE \\
\textbf{channel}\ apex\_req\_partition\_status : PARTITION\_ID\_TYPE \\
\textbf{channel}\ apex\_get\_partition\_status : PARTITION\_ID\_TYPE \\
\qquad \times PARTITION\_STATUS\_TYPE \\
\textbf{channel}\ apex\_req\_system\_time : PARTITION\_ID\_TYPE \\
\textbf{channel}\ apex\_get\_system\_time : PARTITION\_ID\_TYPE \\
\qquad \times SYSTEM\_TIME\_TYPE \\
\textbf{channel}\ apex\_req\_partition\_mode : PARTITION\_ID\_TYPE \\
\textbf{channel}\ apex\_get\_partition\_mode : PARTITION\_ID\_TYPE \\
\qquad \times OPERATING\_MODE\_TYPE \\
\textbf{channel}\ apex\_new\_partition\_mode\_fail : PARTITION\_ID\_TYPE
\end{array}
$$

4. Channels for communication between the APEX and its health monitor

$$
\begin{array}{l}
\textbf{channel}\ apex\_send\_mpHM\_table : iseq_1[\,A653\_MultiPartitionHMTableType\,] \\
\qquad \times (\mathrm{seq}_1\ A653\_ErrorIdentifierType)
\end{array}
$$

5. Channels for communication between Partitions and the APEX

$$
\begin{array}{l}
\textbf{channel}\ part\_set\_partition\_mode : PARTITION\_ID\_TYPE \\
\qquad \times ARINC\_CONSTANTS \\
\textbf{channel}\ part\_req\_partition\_status : PARTITION\_ID\_TYPE \\
\textbf{channel}\ part\_get\_partition\_status : PARTITION\_ID\_TYPE \\
\qquad \times PARTITION\_STATUS\_TYPE \\
\textbf{channel}\ part\_req\_system\_time : PARTITION\_ID\_TYPE \\
\textbf{channel}\ part\_get\_system\_time : PARTITION\_ID\_TYPE \\
\qquad \times SYSTEM\_TIME\_TYPE \\
\textbf{channel}\ part\_req\_sampling\_port\_id : PARTITION\_ID\_TYPE \\
\qquad \times SAMPLING\_PORT\_NAME\_TYPE \\
\textbf{channel}\ part\_get\_sampling\_port\_id : PARTITION\_ID\_TYPE \\
\qquad \times SAMPLING\_PORT\_ID\_TYPE \\
\textbf{channel}\ partHM\_restart\_partition : PARTITION\_ID\_TYPE
\end{array}
$$

6. Channels for communication between each Partition and its health monitor

$$
\begin{aligned}
&\textbf{channel}\, \mathit{part\_send\_partHM\_table} : \mathit{PARTITION\_ID\_TYPE}\\
&\qquad \times \mathit{A653\_PartitionHMTableType}\\
&\qquad \times (\mathrm{seq}_1\ \mathit{A653\_ErrorIdentifierType})
\end{aligned}
$$

7. Channels for communication between ARINC processes and its Partition

$$
\begin{aligned}
&\textbf{channel}\, \mathit{proc\_req\_system\_time} : \mathit{PARTITION\_ID\_TYPE}\\
&\qquad \times \mathit{PROCESS\_ID\_TYPE}\\
&\textbf{channel}\, \mathit{proc\_get\_system\_time} : \mathit{PARTITION\_ID\_TYPE}\\
&\qquad \times \mathit{PROCESS\_ID\_TYPE}\\
&\qquad \times \mathit{SYSTEM\_TIME\_TYPE}\\
&\textbf{channel}\, \mathit{proc\_set\_partition\_mode} : \mathit{PARTITION\_ID\_TYPE}\\
&\qquad \times \mathit{PROCESS\_ID\_TYPE}\\
&\qquad \times \mathit{ARINC\_CONSTANTS}\\
&\textbf{channel}\, \mathit{proc\_req\_sampling\_port\_id} : \mathit{PARTITION\_ID\_TYPE}\\
&\qquad \times \mathit{PROCESS\_ID\_TYPE}\\
&\qquad \times \mathit{SAMPLING\_PORT\_NAME\_TYPE}\\
&\textbf{channel}\, \mathit{proc\_get\_sampling\_port\_id} : \mathit{PARTITION\_ID\_TYPE}\\
&\qquad \times \mathit{PROCESS\_ID\_TYPE}\\
&\qquad \times \mathit{SAMPLING\_PORT\_ID\_TYPE}\\
&\textbf{channel}\, \mathit{proc\_req\_partition\_status} : \mathit{PARTITION\_ID\_TYPE}\\
&\qquad \times \mathit{PROCESS\_ID\_TYPE}\\
&\textbf{channel}\, \mathit{proc\_get\_partition\_status} : \mathit{PARTITION\_ID\_TYPE}\\
&\qquad \times \mathit{PROCESS\_ID\_TYPE}\\
&\qquad \times \mathit{PARTITION\_STATUS\_TYPE}
\end{aligned}
$$

8. Channel for communication between the Operating System and its health monitor

$$
\begin{aligned}
&\textbf{channel}\, \mathit{os\_send\_modHM\_table} : \mathit{iseq}_1[\,\mathit{A653\_ModuleHMTableType}\,]\\
&\qquad \times \mathrm{seq}_1\ \mathit{A653\_ErrorIdentifierType}
\end{aligned}
$$

9. Channel for communication between the Timer and the Operating System

$$
\textbf{channel}\, \mathit{updateClock} : \mathit{SYSTEM\_TIME\_TYPE}
$$

A.3 Partitions

A.3.1 Channels

$$\begin{array}{l}\textbf{channelset}\ PartitionApex == \{\!|\ part\_req\_partition\_status, part\_get\_partition\_status,\\ \quad part\_req\_sampling\_port\_id, part\_get\_sampling\_port\_id,\\ \quad part\_req\_system\_time, part\_get\_system\_time,\\ \quad return\_code, part\_set\_partition\_mode,\\ \quad moduleInit, moduleEnd, moduleEndInit, endPartition, reinitPartition\ |\!\}\end{array}$$

A.3.2 Partition process

$$\begin{array}{l}\textbf{process}\ \ A653\_Partition\ \widehat{=}\ partitionId : PARTITION\_ID\_TYPE;\\ \quad partHM : A653\_PartitionHMTableType;\\ \qquad sysError : \mathrm{seq}_1\ A653\_ErrorIdentifierType\ \bullet\ \textbf{begin}\end{array}$$

$$\begin{array}{l}GetTime\ \widehat{=}\\ \quad proc\_req\_system\_time.partitionId?processId \longrightarrow\\ \quad part\_req\_system\_time.partitionId \longrightarrow\\ \quad part\_get\_system\_time.partitionId?t \longrightarrow\\ \quad proc\_get\_system\_time.partitionId.processId!t \longrightarrow\\ \quad return\_code.partitionId?rc \longrightarrow \textbf{Skip}\end{array}$$

$$\begin{array}{l}SetPartitionMode\ \widehat{=}\\ \quad proc\_set\_partition\_mode.partitionId?processId?OPERATING\_MODE \longrightarrow\\ \quad part\_set\_partition\_mode.partitionId!OPERATING\_MODE \longrightarrow\\ \quad return\_code.partitionId?rc \longrightarrow \textbf{Skip}\end{array}$$

$$\begin{array}{l}GetSamplingPortId\ \widehat{=}\\ \quad proc\_req\_sampling\_port\_id.partitionId?processId?SAMPLING\_PORT\_NAME \longrightarrow\\ \quad part\_req\_sampling\_port\_id.partitionId!SAMPLING\_PORT\_NAME \longrightarrow\\ \quad part\_get\_sampling\_port\_id.partitionId?spid \longrightarrow\\ \quad proc\_get\_sampling\_port\_id.partitionId!processId!spid \longrightarrow\\ \quad return\_code.partitionId?rc \longrightarrow \textbf{Skip}\end{array}$$

$$\begin{array}{l} GetPartitionStatus \mathrel{\widehat=} \\ \quad proc\_req\_partition\_status.partitionId?processId \longrightarrow \\ \quad part\_req\_partition\_status.partitionId \longrightarrow \\ \quad part\_get\_partition\_status.partitionId?st \longrightarrow \\ \quad proc\_get\_partition\_status.partitionId.processId!st \longrightarrow \\ \quad return\_code.partitionId?rc \longrightarrow \mathbf{Skip} \end{array}$$

$$\begin{array}{l} ExecPartitionServices \mathrel{\widehat=} \\ \quad execPartition.partitionId \longrightarrow \\ \quad \left(\begin{array}{l} \mu X \bullet \left(\begin{array}{l} SetPartitionMode \\ \Box\ GetPartitionStatus \\ \Box\ GetSamplingPortId \\ \Box\ GetTime \end{array} \right) ;\ X \\ \triangle\ (interruptPartition.partitionId \longrightarrow \mathbf{Skip}) \end{array} \right) \end{array}$$

$$\begin{array}{l} ExecPartition \mathrel{\widehat=} initPartition.partitionId \longrightarrow \\ \quad part\_send\_partHM\_table.partitionId!pHM!sysError \longrightarrow \\ \quad moduleEndInit \longrightarrow \\ \quad \left(\begin{array}{l} (\mu X \bullet ExecPartitionServices ;\ X) \\ \triangle \left(\begin{array}{l} endPartition.partitionId \longrightarrow \mathbf{Skip} \\ \Box\ reinitPartition.partitionId \longrightarrow ReinitPartition \end{array} \right) \end{array} \right) \end{array}$$

$$\begin{array}{l} ReinitPartition \mathrel{\widehat=} partHM\_restart\_partition.partitionId \longrightarrow \\ \quad \left(\begin{array}{l} (\mu X \bullet ExecPartitionServices ;\ X) \\ \triangle \left(\begin{array}{l} endPartition.partitionId \longrightarrow \mathbf{Skip} \\ \Box\ reinitPartition.partitionId \longrightarrow ReinitPartition \end{array} \right) \end{array} \right) \end{array}$$

$\bullet\ moduleInit \longrightarrow (ExecPartition \triangle\ moduleEnd \longrightarrow \mathbf{Skip})$

end

A.3.3 The Partition Level Health Monitor

process $PartitionHM \mathrel{\widehat=} partitionId : PARTITION\_ID\_TYPE \bullet$ **begin**

$$
\begin{array}{l}
\quad\textbf{state}\ PartitionHMSt == [pHM : A653\_PartitionHMTableType;\\
\qquad\qquad sysError : \mathrm{seq}_1\ A653\_ErrorIdentifierType]\\[1ex]
\quad\bullet \left(\begin{array}{l} part\_send\_partHM\_table.partitionId?phm?se \longrightarrow\\ pHM, sysError := phm, se\end{array}\right)\\
\qquad\qquad \triangle \left(\begin{array}{l} moduleEnd \longrightarrow \mathbf{Skip}\\ \Box\ endPartition.partitionId \longrightarrow \mathbf{Skip}\end{array}\right)
\end{array}
$$

end

$$
\begin{array}{l}
\textbf{channelset}\ PartitionHMChannels ==\\
\qquad\qquad \{\!|\ part\_send\_partHM\_table, moduleEnd, endPartition\ |\!\}
\end{array}
$$

A.3.4 Partitions Layer Model

$$
\begin{array}{l}
\textbf{process}\ \ Partitions\_Layer\ \widehat{=}\ partitionIds : \mathbb{P}\ PARTITION\_ID\_TYPE;\\
\quad partHM : \mathrm{seq}_1\ A653\_PartitionHMTableType;\\
\qquad sysError : \mathrm{seq}_1\ A653\_ErrorIdentifierType\\
\qquad \bullet \left(\begin{array}{l} \|\ pid : partitionIds\ [\![\ \{\!|\ moduleInit, moduleEnd, moduleEndInit\ |\!\}\ ]\!]\\ \bullet \left(\begin{array}{l} A653\_Partition(pid, partHM(pid), sysError)\\ \qquad [\![PartitionHMChannels\ ]\!]\ PartitionHM(pid)\end{array}\right)\end{array}\right)
\end{array}
$$

A.4 The APEX

$$
\begin{array}{l}
\textbf{channelset}\ ApexOs == \{\!|\ moduleInit, moduleEnd, moduleEndInit,\\
\qquad endPartition, reinitPartition, return\_code,\\
\qquad apex\_get\_sampling\_port\_id, apex\_req\_new\_partition\_mode,\\
\qquad apex\_req\_partition\_status, apex\_get\_partition\_status,\\
\qquad apex\_req\_system\_time, apex\_get\_system\_time,\\
\qquad apex\_req\_partition\_mode, apex\_get\_partition\_mode,\\
\qquad apex\_req\_sampling\_port\_id, apex\_new\_partition\_mode\_fail\ |\!\}
\end{array}
$$

A.4.1 APEX Process

$$
\begin{array}{l}
\textbf{process}\ \ APEX\ \widehat{=}\ sysError : \mathrm{seq}_1\ A653\_ErrorIdentifierType;\\
\qquad\qquad\qquad mpHM : \mathrm{seq}_1\ A653\_MultiPartitionHMTableType \bullet \textbf{begin}
\end{array}
$$

$$
\begin{array}{l}
GET\_TIME \mathrel{\widehat=} part\_req\_system\_time?pid \longrightarrow \\
\quad apex\_req\_system\_time.pid \longrightarrow \\
\quad apex\_get\_system\_time.pid?t \longrightarrow \\
\quad part\_get\_system\_time.pid!t \longrightarrow \\
\quad return\_code.pid!NO\_ERROR \longrightarrow \mathbf{Skip}
\end{array}
$$

$$
\begin{array}{l}
GET\_PARTITION\_STATUS \mathrel{\widehat=} \\
\quad part\_req\_partition\_status?pid \longrightarrow \\
\quad apex\_req\_partition\_status.pid \longrightarrow \\
\quad apex\_get\_partition\_status.pid?cp \longrightarrow \\
\quad part\_get\_partition\_status.pid!cp \longrightarrow \mathbf{Skip}
\end{array}
$$

$$
\begin{array}{l}
SET\_PARTITION\_MODE \mathrel{\widehat=} \\
\quad part\_set\_partition\_mode?pid?nopm \longrightarrow \\
\quad apex\_req\_partition\_mode.pid \longrightarrow \\
\quad apex\_get\_partition\_mode.pid?opm \longrightarrow \\
\qquad \left(\begin{array}{l}
\mathbf{if}\ (nopm \notin OPERATING\_MODE\_TYPE) \longrightarrow \\
\qquad return\_code.pid!INVALID\_PARAM \longrightarrow \mathbf{Skip} \\
[\!]\ (opm = NORMAL \land nopm = NORMAL) \longrightarrow \\
\qquad apex\_new\_partition\_mode\_fail.pid \longrightarrow \\
\qquad return\_code.pid!NO\_ACTION \longrightarrow \mathbf{Skip} \\
[\!]\ (opm = COLD\_START \land nopm = WARM\_START) \longrightarrow \\
\qquad apex\_new\_partition\_mode\_fail.pid \longrightarrow \\
\qquad return\_code.pid!INVALID\_MODE \longrightarrow \mathbf{Skip} \\
[\!]\ \left(\begin{array}{l} nopm = IDLE \\ \lor\ nopm \in OPERATING\_INIT\_MODE\_TYPE \end{array}\right) \longrightarrow \\
\qquad apex\_req\_new\_partition\_mode.pid!nopm \longrightarrow \\
\qquad return\_code.pid!NO\_ERROR \longrightarrow \mathbf{Skip} \\
[\!]\ (opm \neq NORMAL \land nopm = NORMAL) \longrightarrow \\
\qquad apex\_req\_new\_partition\_mode.pid!nopm \longrightarrow \\
\qquad return\_code.pid!NO\_ERROR \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
GET\_SAMPLING\_PORT\_ID \mathrel{\widehat=} \\
\quad part\_req\_sampling\_port\_id?pid?spn \longrightarrow \\
\quad apex\_req\_sampling\_port\_id.pid!spn \longrightarrow \\
\quad apex\_get\_sampling\_port\_id.pid?spid \longrightarrow \\
\qquad \left(\begin{array}{l}
\mathbf{if}\ (spid = \text{-}1) \longrightarrow \\
\qquad part\_get\_sampling\_port\_id.pid!spid \longrightarrow \\
\qquad return\_code.pid!INVALID\_CONFIG \longrightarrow \mathbf{Skip} \\
[\!]\ (spid \neq \text{-}1) \longrightarrow \\
\qquad part\_get\_sampling\_port\_id.pid!spid \longrightarrow \\
\qquad return\_code.pid!NO\_ERROR \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
ExecApexServices \mathrel{\widehat=} execPartition?pid \longrightarrow \\
\quad \left(\begin{array}{l}
\mu X \bullet \left(\begin{array}{l}
GET\_TIME \\
\Box\ SET\_PARTITION\_MODE \\
\Box\ GET\_PARTITION\_STATUS \\
\Box\ GET\_SAMPLING\_PORT\_ID
\end{array}\right) ;\ X \\
\triangle \left(\begin{array}{l}
interruptPartition.pid \longrightarrow \mathbf{Skip} \\
\Box\ endPartition.pid \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
ExecApex \mathrel{\widehat=} apex\_send\_mpHM\_table!mpHM!sysError \longrightarrow \\
\quad moduleEndInit \longrightarrow (\mu X \bullet ExecApexServices\ ;\ X)
\end{array}
$$

$$\bullet\ moduleInit \longrightarrow (ExecApex \triangle\ moduleEnd \longrightarrow \mathbf{Skip})$$

end

A.4.2 The APEX Health Monitor (Multi-Partition)

process $MultiPartitionHM \mathrel{\widehat=}$ **begin**

$$
\begin{array}{l}
\mathbf{state}\ MultiPartitionHMSt == \\
\qquad [mpHM : \mathrm{seq}_1\ A653\_MultiPartitionHMTableType; \\
\qquad\quad sysError : \mathrm{seq}_1\ A653\_ErrorIdentifierType] \\
\bullet\ apex\_send\_mpHM\_table?mp?se \longrightarrow \\
\qquad mpHM, sysError := mp, se \\
\qquad\quad \triangle moduleEnd \longrightarrow \mathbf{Skip}
\end{array}
$$

end

$\textbf{channelset}\ \mathit{MultiPartitionHMChannels} == \{\!|\ \mathit{apex\_send\_mpHM\_table}, \mathit{moduleEnd}\ |\!\}$

A.4.3 APEX Layer Model

$$\begin{array}{l}
\textbf{process}\ \mathit{APEX\_Layer} \mathrel{\widehat=} \mathit{mpHM} : \mathrm{seq}_1\ \mathit{A653\_MultiPartitionHMTableType}; \\
\qquad \mathit{sysError} : \mathrm{seq}_1\ \mathit{A653\_ErrorIdentifierType} \\
\qquad\quad \bullet \left(\begin{array}{l} \mathit{APEX}(\mathit{sysError}, \mathit{mpHM}) \\ \qquad [\![\mathit{MultiPartitionHMChannels}]\!]\ \mathit{MultiPartitionHM} \end{array}\right)
\end{array}$$

A.5 The Operating System

A.5.1 The Operating System *Circus* process

$\textbf{process}\ \mathit{OperatingSystem} \mathrel{\widehat=} \mathit{module} : \mathit{Module} \bullet \textbf{begin}$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{OSSt}} \\
\hline
\mathit{partitions\_variables} : \mathit{iseq}_1[\mathit{PartitionVariables}] \\
\mathit{major\_time\_frame} : \mathit{SYSTEM\_TIME\_TYPE}; \\
\mathit{system\_time} : \mathit{SYSTEM\_TIME\_TYPE} \\
\mathit{current\_partition} : \mathit{A653\_PartitionTimeWindowType} \\
\mathit{current\_partition\_id} : \mathit{PARTITION\_ID\_TYPE} \\
\mathit{sampling\_ports} : (\mathit{SAMPLING\_PORT\_ID\_TYPE} \rightarrowtail\!\!\!\!\!+ \\
\qquad\qquad\qquad \mathit{SAMPLING\_PORT\_NAME\_TYPE}) \\
\mathit{queuing\_ports} : (\mathit{QUEUING\_PORT\_ID\_TYPE} \rightarrowtail\!\!\!\!\!+ \\
\qquad\qquad\qquad \mathit{QUEUING\_PORT\_NAME\_TYPE}) \\
\hline
\#\,\mathit{partitions\_variables} = \#\,\mathit{module}.\mathit{Partitions} \\
\hline
\end{array}$$

$\textbf{state}\ \mathit{OSSt}$

$\mathit{InitPartition} \mathrel{\widehat=} |||\ \mathit{pid} : (1 \mathbin{..} (\#\,\mathit{module}.\mathit{Partitions})) \bullet \mathit{initPartition}.\mathit{pid} \longrightarrow \textbf{Skip}$

$$
\begin{array}{|l}
\hline
\textit{MajorTimeFrame} \\
\Delta \mathit{OSSt} \\
\mathit{getMajorTimeFrame} : \mathit{PartitionsType} \\
\hline
\forall x : \mathrm{ran}\ \mathit{module}.\mathit{Partitions} \\
\quad \bullet \# (\mathit{module}.\mathit{Partitions} \rhd \{x\}) = \# (\mathit{getMajorTimeFrame} \rhd \{x\}) \\
\forall p1, p2 : \mathrm{ran}\ \mathit{getMajorTimeFrame} \\
\quad \bullet p1.\mathit{PartitionPeriodicity}.\mathit{Period} > p2.\mathit{PartitionPeriodicity}.\mathit{Period} \\
\mathit{major\_time\_frame}' = (\mathit{getMajorTimeFrame}(1)).\mathit{PartitionPeriodicity}.\mathit{Period} \\
\theta(\mathit{OSSt} \setminus (\mathit{major\_time\_frame}))' = \theta(\mathit{OSSt} \setminus (\mathit{major\_time\_frame})) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{NextPartition} \\
\Delta \mathit{OSSt} \\
\hline
\mathit{current\_partition}' = \mathit{head}\ (\mathit{module}.\mathit{Schedules} \upharpoonright \\
\qquad \{p : \mathit{A653\_PartitionTimeWindowType} \mid \\
\qquad\qquad (\mathit{system\_time} \bmod \mathit{major\_time\_frame}) \leq p.\mathit{Offset}\}) \\
(\mathit{module}.\mathit{Partitions}(\mathit{current\_partition\_id}')).\mathit{PartitionDefinition}.\mathit{Name} = \\
\qquad\qquad (\mathit{current\_partition}').\mathit{PartitionNameRef} \\
\theta(\mathit{OSSt} \setminus (\mathit{current\_partition\_id}, \mathit{current\_partition}))' = \\
\quad \theta(\mathit{OSSt} \setminus (\mathit{current\_partition\_id}, \mathit{current\_partition})) \\
\hline
\end{array}
$$

$$\mathit{UpdateSystemTime} \mathrel{\widehat=} \mathit{updateClock}?x \longrightarrow \mathit{system\_time} := x$$

$$\begin{array}{l}\mathit{OSGetTime} \mathrel{\widehat=} \mathit{apex\_req\_system\_time}.\mathit{current\_partition\_id} \longrightarrow \mathit{UpdateSystemTime}; \\ \qquad \mathit{apex\_get\_system\_time}.\mathit{current\_partition\_id}!\mathit{system\_time} \longrightarrow \mathbf{Skip}\end{array}$$

$$
\begin{array}{l}
\textit{PartitionStatus} \\
\hline
\Xi \textit{OSSt} \\
p! : \textit{PARTITION\_STATUS\_TYPE} \\
\hline
(p!).\textit{Identifier} = \\
\quad (\textit{module}.\textit{Partitions}(\textit{current\_partition\_id})).\textit{PartitionDefinition}.\textit{Identifier} \\
(p!).\textit{Period} = \\
\quad (\textit{module}.\textit{Partitions}(\textit{current\_partition\_id})).\textit{PartitionPeriodicity}.\textit{Period} \\
(p!).\textit{Duration} = \\
\quad (\textit{module}.\textit{Partitions}(\textit{current\_partition\_id})).\textit{PartitionPeriodicity}.\textit{Duration} \\
(p!).\textit{LOCK\_LEVEL} = \\
\quad (\textit{partitions\_variables}(\textit{current\_partition\_id})).\textit{LOCK\_LEVEL} \\
(p!).\textit{OPERATING\_MODE} = \\
\quad (\textit{partitions\_variables}(\textit{current\_partition\_id})).\textit{OPERATING\_MODE} \\
(p!).\textit{START\_CONDITION} = \\
\quad (\textit{partitions\_variables}(\textit{current\_partition\_id})).\textit{START\_CONDITION} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{OSGetPartitionStatus} \mathrel{\widehat=} \mathbf{var}\, p : \textit{PARTITION\_STATUS\_TYPE} \bullet \\
\quad \textit{apex\_req\_partition\_status}.\textit{current\_partition\_id} \longrightarrow \big(\textit{PartitionStatus}\big); \\
\quad\quad \textit{apex\_get\_partition\_status}.\textit{current\_partition\_id}!p \longrightarrow \mathbf{Skip}
\end{array}
$$

$$
\begin{array}{l}
\textit{SetPMode} \\
\hline
\Delta \textit{OSSt} \\
\textit{nopm}? : \textit{OPERATING\_MODE\_TYPE} \\
\hline
\textit{partitions\_variables}' = \\
(\mathbf{let}\ o == \langle\!|\ \textit{OPERATING\_MODE} == \textit{nopm}?, \\
\textit{START\_CONDITION} == \\
\quad (\textit{partitions\_variables}(\textit{current\_partition\_id})).\textit{START\_CONDITION}, \\
\textit{LOCK\_LEVEL} == \\
\quad (\textit{partitions\_variables}(\textit{current\_partition\_id})).\textit{LOCK\_LEVEL}\ |\!\rangle \\
\quad \bullet\ \textit{partitions\_variables} \oplus \{\textit{current\_partition\_id} \mapsto o\}) \\
\theta(\textit{OSSt} \setminus (\textit{current\_partition\_id}, \textit{current\_partition}, \textit{partitions\_variables}))' = \\
\quad \theta(\textit{OSSt} \setminus (\textit{current\_partition\_id}, \textit{current\_partition}, \textit{partitions\_variables})) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
OSSetPartitionMode \mathrel{\widehat=} \mathbf{var}\ opm : OPERATING\_MODE\_TYPE \bullet \\
\quad apex\_req\_partition\_mode.current\_partition\_id \longrightarrow \\
\quad apex\_get\_partition\_mode.current\_partition\_id!(\\
\quad (PartitionsVariables(current\_partition\_id)).OPERATING\_MODE) \longrightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
apex\_req\_new\_partition\_mode.current\_partition\_id?nopm \longrightarrow \\
\mathbf{if}(nopm = IDLE) \longrightarrow \\
\qquad endPartition.current\_partition\_id \longrightarrow \big(SetPMode\big) \\
[\!]\ (nopm \in OPERATING\_INIT\_MODE\_TYPE) \longrightarrow \\
\qquad reinitPartition.current\_partition\_id \longrightarrow \big(SetPMode\big) \\
[\!]\ (nopm = NORMAL) \longrightarrow \big(SetPMode\big) \\
\mathbf{fi}
\end{array}\right) \\
\Box\ apex\_new\_partition\_mode\_fail?current\_partition\_id \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
OSGetSamplingPortId \mathrel{\widehat=} \\
\quad apex\_req\_sampling\_port\_id.current\_partition\_id?spn \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}(spn \in \operatorname{ran} sampling\_ports) \longrightarrow \\
\qquad apex\_get\_sampling\_port\_id.current\_partition\_id!(\mu \\
\qquad\qquad\qquad\qquad x : (\operatorname{dom}(sampling\_ports \rhd \{spn\})) \bullet x) \longrightarrow \\
\qquad return\_code.current\_partition\_id!NO\_ERROR \longrightarrow \mathbf{Skip} \\
[\!]\ (spn \notin \operatorname{ran} sampling\_ports) \longrightarrow \\
\qquad apex\_get\_sampling\_port\_id.current\_partition\_id!(\text{-}\,1) \longrightarrow \\
\qquad return\_code.current\_partition\_id!INVALID\_CONFIG \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
ExecOSServices \mathrel{\widehat=} \\
\quad execPartition.current\_partition\_id \longrightarrow \\
\quad \left(\mu X \bullet \left(\begin{array}{l}
OSGetTime \\
\Box\ OSGetPartitionStatus \\
\Box\ OSSetPartitionMode \\
\Box\ OSGetSamplingPortId
\end{array}\right) ;\ X \right)
\end{array}
$$

$$
\begin{array}{l}
\mathit{ExecOS} \mathrel{\widehat=} \mathbf{var}\ \mathit{getMajorTimeFrame} : \mathit{PartitionsType} \bullet \big(\mathit{MajorTimeFrame}\big); \\
\left(\begin{array}{l}
\mu X \bullet \mathit{UpdateSystemTime} ;\ \big(\mathit{NextPartition}\big); \\
\left(\begin{array}{l}
\mathbf{if} \left(\begin{array}{l} \mathit{current\_partition.Offset} > \\ \qquad (\mathit{system\_time} \bmod \mathit{major\_time\_frame}) \end{array}\right) \longrightarrow \\
\qquad \mathbf{wait} \left(\begin{array}{l} \mathit{current\_partition.Offset} - \\ \qquad (\mathit{system\_time} \bmod \mathit{major\_time\_frame}) \end{array}\right) ;\ X \\
[\!] \left(\begin{array}{l} \mathit{current\_partition.Offset} = \\ \qquad (\mathit{system\_time} \bmod \mathit{major\_time\_frame}) \end{array}\right) \longrightarrow \\
\qquad \left(\begin{array}{l} \mathit{ExecOSServices} \overset{\mathit{current\_partition.Duration}}{\rhd} \\ \qquad \mathit{interruptPartition.current\_partition\_id} ;\ X \end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
\bullet\ \mathit{moduleInit} \longrightarrow \\
\quad \mathit{os\_send\_modHM\_table}!(\mathit{module.HealthMonitoring.ModuleHM})!(\\
\qquad \mathit{module.HealthMonitoring.SystemErrors}) \longrightarrow (\mathit{ExecOS} \\
\qquad\quad \triangle\ \mathit{moduleEnd} \longrightarrow \mathbf{Skip})
\end{array}
$$

end

A.5.2 The Timer

process *Timer* $\widehat=$ **begin**

$$\mathbf{state}\ \mathit{TimerSt} == [\mathit{clock} : \mathit{SYSTEM\_TIME\_TYPE}]$$

$$
\mathit{Counter} \mathrel{\widehat=} \left(\begin{array}{l}
\mu X \bullet \\
\left(\begin{array}{l} (\mathbf{wait}(1) ;\ \mathit{clock} := \mathit{clock} + 1) \\ ||| \left((\mu Y \bullet \mathit{updateClock}!\mathit{clock} \longrightarrow Y) \overset{1}{\rhd} \mathbf{Skip} \right) \end{array}\right) ;\ X
\end{array}\right)
$$

$$\bullet\ \mathit{moduleInit} \longrightarrow (\mathit{Counter}\ \triangle\ \mathit{moduleEnd} \longrightarrow \mathbf{Skip})$$

end

A.5.3 The Operating System Health Monitor (Module)

process *ModuleHM* $\widehat=$ **begin**

$$\begin{array}{l}
\textbf{state}\ \ \mathit{ModuleHMSt} == [\mathit{modHM} : \mathrm{seq}_1\ \mathit{A653\_ModuleHMTableType};\\
\qquad\qquad\qquad \mathit{sysError} : \mathrm{seq}_1\ \mathit{A653\_ErrorIdentifierType}]\\
\bullet\ (\mathit{os\_send\_modHM\_table}?\mathit{mp}?\mathit{se} \longrightarrow \mathit{modHM}, \mathit{sysError} := \mathit{mp}, \mathit{se})\\
\qquad\qquad\qquad \triangle \mathit{moduleEnd} \longrightarrow \mathbf{Skip}
\end{array}$$

end

$$\textbf{channelset}\ \mathit{ModuleHMChannels} == \{\!|\ \mathit{os\_send\_modHM\_table}, \mathit{moduleEnd}\ |\!\}$$

A.5.4 The Operating System Layer Model

$$\begin{array}{l}
\textbf{process}\ \ \mathit{OS\_Layer}\ \widehat{=}\ \mathit{module} : \mathit{Module}\\
\qquad \bullet \left(\begin{array}{l} \left(\begin{array}{l} \mathit{OperatingSystem}(\mathit{module})\\ [\![\{\!|\ \mathit{updateClock}\ |\!\}]\!]\\ \mathit{Timer} \end{array} \right)\\ [\![\mathit{ModuleHMChannels}\ ]\!]\ \mathit{ModuleHM} \end{array} \right)
\end{array}$$

$$\begin{array}{l}
\textbf{process}\ \ \mathit{IMA\_Module}\ \widehat{=}\ \mathit{module} : \mathit{Module}\\
\bullet \left(\begin{array}{l} \mathit{OS\_Layer}(\mathit{module})\\ [\![\mathit{ApexOs}]\!]\\ \left(\begin{array}{l} \mathit{APEX\_Layer} \left(\begin{array}{l} \mathit{module}.\mathit{HealthMonitoring}.\mathit{MultiPartitionHM},\\ \mathit{module}.\mathit{HealthMonitoring}.\mathit{SystemErrors} \end{array} \right)\\ [\![\mathit{PartitionApex}]\!]\\ \mathit{Partitions\_Layer} \left(\begin{array}{l} 1 \ldots \#\ \mathit{module}.\mathit{Partitions},\\ \mathit{module}.\mathit{HealthMonitoring}.\mathit{PartitionHM},\\ \mathit{module}.\mathit{HealthMonitoring}.\mathit{SystemErrors} \end{array} \right) \end{array} \right) \end{array} \right)
\end{array}$$

Bibliography

[1] DO-178B: Software Considerations in Airborne Systems and Equipment Certification, 1992.

[2] ARINC 653 - Avionics Application Software Standartd Interface, November 2010.

[3] Jean-Raymond Abrial. *The B-book - Assigning Programs to Meanings*. Cambridge University Press, 2005.

[4] Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors. *Integrated Formal Methods, Proceedings of the 1st International Conference on Integrated Formal Methods, IFM 99, York, UK, 28-29 June 1999*. Springer, 1999.

[5] ARINC. Aeronautical Radio, Incorporated (ARINC), July 2011.

[6] Michael Butler and Michael Leuschel. Combining CSP and B for Specification and Property Verification. In *In Proceedings of Formal Methods 2005 (in press), Newcastle upon*, pages 221–236. Springer, 2005.

[7] A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. From Control Law Diagrams to Ada via *Circus*. *Formal Aspects of Computing*, 23(4):465 – 512, 2011.

[8] A. L. C. Cavalcanti and Phil Clayton. Verification of Control Systems using *Circus*. In *ICECCS '06: Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems*, pages 269–278, Washington, DC, USA, 2006. IEEE Computer Society.

[9] A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC—A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267—289, 1999.

[10] Clearsy. Atelier B. `http://www.atelierb.eu/`, August 2011.

[11] Philippa Conmy, Mark Nicholson, and John McDermid. Safety Assurance Contracts for Integrated Modular Avionics. In *SCS '03: Proceedings of the 8th Australian workshop on Safety critical systems and software*, pages 69–78, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[12] A. Cooka and K.J.R. Hunt. ARINC 653 — Achieving Software Re-use. In *Microprocessors and Microsystems*, volume 20, pages 479–483. Elsevier, 1997.

[13] Julien Delange, Laurent Pautet, and Fabrice Kordon. Modeling and Validation of ARINC653 architectures. In *Embedded Real-time Software and Systems Conference*, 2010.

[14] Julien Delange, Laurent Pautet, Alain Plantec, Mickael Kerboeuf, Frank Singhoff, and Fabrice Kordon. Validate, Simulate, and Implement ARINC653 Systems Using the AADL. *Ada Lett.*, 29:31–44, November 2009.

[15] Ben L. Di Vito. A formal model of partitioning for integrated modular avionics. NASA Contractor Report CR-1998-208703, NASA Langley Research Center, August 1998.

[16] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18:453–457, August 1975.

[17] Roger Duke, Gordon Rose, and Graeme Smith. Object-Z: a Specification Language Advocated for the Description of Standards. *COMPUTER STANDARDS AND INTERFACES*, 17, 1995.

[18] J. Fenn, R. Hawkins, P. Williams, T. Kelly, M. Banner, and Y Oakshott. The Who, Where, How, Why and When of Modular and Incremental Certification. In *2nd IET International Conference on System Safety.* IET, 2007.

[19] Clemens Fischer. CSP-OZ: a Combination of Object-Z and CSP. In *Proceedings of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems*, pages 423–438, London, UK, UK, 1997. Chapman & Hall, Ltd.

[20] John S. Fitzgerald and Peter Gorm Larsen. *Modelling Systems - Practical Tools and Techniques in Software Development (2. ed.).* Cambridge University Press, 2009.

[21] A. J. Galloway and W. J. Stoddart. An Operational Semantics for ZCCS. In *Proceedings of the 1st International Conference on Formal Engineering Methods*, ICFEM '97, pages 272–, Washington, DC, USA, 1997. IEEE Computer Society.

[22] Abdoulaye Gamati, Christian Brunette, Romain Delamare, Thierry Gautier, and Jean-Pierre Talpin. A Modeling Paradigm for Integrated Modular Avionics Design. *EUROMICRO Conference*, 0:134–143, 2006.

[23] Abdoulaye Gamatie and Thierry Gautier. The Signal Synchronous Multiclock Approach to the Design of Distributed Embedded Systems. *IEEE Transactions on Parallel and Distributed Systems*, 99(RapidPosts):641–657, 2009.

[24] Artur Oliveira Gomes. *Circus* specification of the IMA architecture. `http://www-users.cs.york.ac.uk/~artur/circus-spec.zip`, September 2011.

[25] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. POLYCHRONY for System Design. *Journal of Circuits, Systems, and Computers*, 12(3):261–304, 2003.

[26] Jifeng He and C. A. R. Hoare. Unifying Theories of Programming. In Ewa Orlowska and Andrzej Szalas, editors, *RelMiCS*, pages 97–99, 1998.

[27] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[28] Paul Hollow, John Mcdermid, and Mark Nicholson. Approaches to Certification of Reconfigurable IMA Systems. In *INCOSE 2000*, pages 17–20, 2000.

[29] LISyC laboratory Université de Bretagne Occidentale and Ellidiss Technologies. The Cheddar project : a free real time scheduling analyzer, August 2011.

[30] Formal Systems (Europe) Ltd. Failures-Divergence Refinement: FDR2 User Manual., October 2010.

[31] Formal Systems (Europe) Ltd. ProBE Manual., October 2010.

[32] Lemma 1 Ltd. ProofPower. `http://www.lemma-one.com/ProofPower/`, August 2009.

[33] Ian MacColl and David A. Carrington. Specifying Interactive Systems in Object-Z and CSP. In Araki et al. [4], pages 335–352.

[34] Savi Maharaj and J. Bicarregui. On the Verification of VDM Specification and Refinement with PVS. In *Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*, ASE '97, pages 280–, Washington, DC, USA, 1997. IEEE Computer Society.

[35] Brendan Mahony and Jin Song Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *The 20th International Conference on Software Engineering(ICSE98*, pages 95–104. IEEE Press, 1997.

[36] Chris Marriott. A Tool Chain for the Automatic Generation of *Circus* Specifications from Control Law Diagrams. Master's thesis, University of York, United Kingdom, 2010.

[37] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[38] Carroll C. Morgan. *Programming From Specifications, 2nd Edition.* Prentice Hall International series in computer science. Prentice Hall, 1994.

[39] Alexandre Mota and Augusto Sampaio. Model-Checking CSP-Z. In *In Proceedings of the European Joint Conference on Theory and Practice of Software, volume 1382 of LNCS*, pages 205–220. Springer-Verlag, 1998.

[40] Paul Mukherjee. System Refinement in VDM-SL. In *Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '96, pages 483–, Washington, DC, USA, 1996. IEEE Computer Society.

[41] Mark Nicholson, Philippa Conmy, Iain Bate, and John McDermid. Generating and Maintaining a Safety Argument for Integrated Modular Systems. In *Adelard for the Health and Safety Executive, HSE Books, ISBN 0-7176-2010-7, and Contract Research*, pages 0–7176, 2000.

[42] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science - University of York, UK, 2005. YCST-2006-02.

[43] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[44] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[45] Marie-Laure Potet and Yann Rouzaud. Composition and Refinement in the B-Method. In *Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, pages 46–65, London, UK, 1998. Springer-Verlag.

[46] Pedro Fernando Ribeiro. Modelling Ada implementations of control law diagrams in *Circus*. Master's thesis, University of York, United Kingdom, 2011.

[47] Ken Robinson. The B Method and the B Toolkit. In *Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology*, AMAST '97, pages 576–580, London, UK, 1997. Springer-Verlag.

[48] A. W. Roscoe, Paul H. B. Gardiner, Michael Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking csp or how to check 10^{20} dining philosophers for deadlock. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *TACAS*, volume 1019 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 1995.

[49] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[50] A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-interference through Determinism. In *Journal of Computer Security*, pages 33–53. Springer-Verlag, 1994.

[51] A.W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[52] Ana-Elena Rugina, Peter H. Feiler, Karama Kanoun, and Mohamed Kaâniche. Software dependability modeling using an industry-standard architecture description language. *CoRR*, abs/0809.4109, 2008.

[53] John Rushby. Modular Certification. Technical report, SRI International, 2002.

[54] Mark Saaltink. The Z/EVES System. In *Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation*, ZUM '97, pages 72–85, London, UK, 1997. Springer-Verlag.

[55] Adnan Sherif. *A Framework for Specification and Validation of Real-Time Systems using Circus Actions.* PhD thesis, Center of Informatics - Federal University of Pernambuco, Brazil, 2006.

[56] Adnan Sherif, Ana Cavalcanti, Jifeng He, and Augusto Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Asp. Comput.*, 22(2):153–191, 2010.

[57] Adnan Sherif and Jifeng He. Towards a time model for circus. In Chris George and Huaikou Miao, editors, *ICFEM*, volume 2495 of *Lecture Notes in Computer Science*, pages 613–624. Springer, 2002.

[58] Graeme Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In *Proceedings of FME 1997, volume 1313 of LNCS*, pages 62–81. Springer-Verlag, 1997.

[59] Graeme Smith and John Derrick. Refinement and Verification of Concurrent Systems Specified in Object-Z and CSP. In *First International Conference on Formal Engineering Methods (ICFEM 97*, pages 293–302. IEEE Computer Society Press, 1997.

[60] Graeme Smith and Luke Wildman. Model checking z specifications using sal. In Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 85–103. Springer, 2005.

[61] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal Verification of Avionics Software Products. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 532–546, Berlin, Heidelberg, 2009. Springer-Verlag.

[62] K. Taguchi and K. Araki. The State-Based CCS Semantics for Concurrent Z Specification. In *Proceedings of the 1st International Conference on Formal Engineering Methods*, ICFEM '97, pages 283–, Washington, DC, USA, 1997. IEEE Computer Society.

[63] Helen Treharne and Steve Schneider. Using a Process Algebra to Control B Operations. In Araki et al. [4], pages 437–456.

[64] J. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Series in Computer Science. Prentice Hall International, 1996.

[65] Jim Woodcock and Ana Cavalcanti. The Semantics of Circus. In *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, ZB '02, pages 184–203, London, UK, UK, 2002. Springer-Verlag.

[66] F. Zeyda and A. Cavalcanti. Mechanical Reasoning about Families of UTP Theories. In P. Machado, A. Andrade, and A. Duran, editors, *SBMF 2008, Brazilian Symposium on Formal Methods*, pages 145–160, 2008. Best paper award.