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 Lidiiane 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

1 Introduction
  1.1 Motivation .................................................. 3
  1.2 Objectives .................................................. 7
  1.3 Structure of the dissertation .............................. 7

2 Background and Related Work ..................................... 9
  2.1 Integrated Modular Avionics ............................... 9
    2.1.1 Partitions ............................................. 11
    2.1.2 Schedules ............................................. 12
    2.1.3 Health Monitor ........................................ 13
    2.1.4 Configuration Tables ................................. 13
  2.2 Certification and Verification of IMA systems ............ 15
    2.2.1 Certification Strategies .............................. 15
    2.2.2 Verification of Avionics Systems ..................... 17
  2.3 Formal Specification Languages ............................ 18
  2.4 Circus ...................................................... 19
  2.5 Final Considerations ....................................... 22

3 A Circus Model of the ARINC 653 Components .................. 23
  3.1 Common Types and Channels ............................... 25
    3.1.1 Types ............................................... 25
    3.1.2 Channels ............................................. 28
  3.2 Partitions .................................................. 28
  3.3 The APEX .................................................... 33
  3.4 The Operating System ...................................... 37
    3.4.1 Configuration tables ................................. 37
    3.4.2 Circus process ....................................... 40
    3.4.3 The System Timer ..................................... 45
    3.4.4 The Module Level Health Monitor ..................... 46
  3.5 Translating the Circus model into CSP .................... 48
    3.5.1 Specification of the types for the Configuration Tables 49
    3.5.2 Translation of the A653_Partition Circus process .......... 49
    3.5.3 Translation of Circus processes containing state .......... 50
    3.5.4 The Operating System process in CSP .................. 51
3.6 Validation of the model using FDR ........................................ 56
3.7 Final Considerations ......................................................... 58

4 Conclusions ........................... 61
4.1 Summary and Contributions ............................................. 61
4.2 Future Work ................................................................. 62

A A Circus Model of the ARINC 653 Components .......................... 65
A.1 Types ................................................................. 65
  A.1.1 Basic types definition ............................................ 65
  A.1.2 Configuration Tables Data Structure .......................... 67
A.2 Circus Channels ......................................................... 71
A.3 Partitions ............................................................... 74
  A.3.1 Channels .......................................................... 74
  A.3.2 Partition process .................................................. 74
  A.3.3 The Partition Level Health Monitor ............................ 75
  A.3.4 Partitions Layer Model ........................................... 76
A.4 The APEX ............................................................... 76
  A.4.1 APEX Process ...................................................... 76
  A.4.2 The APEX Health Monitor (Multi-Partition) .................. 78
  A.4.3 APEX Layer Model ................................................ 79
A.5 The Operating System .................................................... 79
  A.5.1 The Operating System Circus process .......................... 79
  A.5.2 The Timer .......................................................... 83
  A.5.3 The Operating System Health Monitor (Module) ............. 83
  A.5.4 The Operating System Layer Model ............................. 84
List of Figures

2.1 ARINC 653 module architecture [2] ............................................. 10
2.2 ARINC 653 - Interpartition communication example ............................. 12
2.3 ARINC 653 - Partition time window example [2] .................................. 13
2.4 ARINC 653 configuration tables structure [2] ........................................ 14

3.1 ARINC 653 - Overview of the channels used in the Circus model of the architecture .......................................................... 23
3.2 ARINC 653 - Channels for the module state transition ............................ 28
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
1.1. MOTIVATION

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 cannot 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.

![Diagram of ARINC 653 module architecture](image)

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](image)

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.

![ARINC 653 - Partition time window example](image)

**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.

![ARINC 653 configuration tables structure](image)

**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 $2 \times 10^7$ 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.
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, degree 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
2.4. **CIRCUS**

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.

\[
\text{[IP_ADDR]}
\]
\[
\text{RESPONSE ::= RESP | TIMEOUT}
\]

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.

\[
\text{channel send_data, receive_data : IP_ADDR}
\]
\[
\text{channel tick}
\]
\[
\text{channel display_resp : RESPONSE \times N}
\]

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

\[
\text{process Ping} \equiv \text{ip : IP_ADDR} \bullet \text{begin}
\]

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 \textbullet symbol. In this example, we present the state PingSt, which contains two components: the maximal number of ping requests is stored in \textit{max_rep}; and the number of requests already performed within the execution of the process is stored in \textit{counter}.

\[
\text{state PingSt} \equiv [\text{max_rep : N};\ \text{counter : N}]
\]

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

\[
\text{InitSt} \equiv \text{max_rep, counter := 4, 0}
\]

The Circus action Send starts the Ping request by sending a signal to the address \textit{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 \textit{ip}.

\[
\text{Send} \equiv \text{send_data}!\text{ip} \rightarrow \text{tick} \rightarrow \text{Receive}(1)
\]
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.

\[
\text{Receive} \equiv \text{time} : 
\begin{align*}
&\text{receive} \cdot \text{data} \cdot ? \cdot \text{ip} \rightarrow \text{display} \cdot \text{resp} \cdot ! \cdot (\text{REC}, \text{time}) \rightarrow \text{counter} := \text{counter} + 1 \\
&\square \cdot \text{tick} \rightarrow \text{Receive}(\text{time} + 1)
\end{align*}
\]

We define the behaviour of the Ping requests by defining the Circus action, \( \text{Run} \) which is a recursion. We formalise a recursion in Circus by using the constructs \( \mu X \cdot P ; X \), where \( P \) is the action being executed. Two behaviours are expected during the recursion, which are specified in an if \( P \rightarrow Q \parallel R \rightarrow S \) 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 \( \text{Run} \) process are the following. (1) If the number of requests has not been reached, \( \text{counter} < \text{max} \cdot \text{rep} \), the Send Circus action is executed with a maximum execution time of 1000 time units, and will be timed out after reaching this interval (\( \triangleright \)). Then, after the timeout of the Send action, the Run Circus action checks if the size of the sequence \( \text{req} \cdot \text{packets} \) is equal to the number of ping requests. If the size of \( \text{req} \cdot \text{packets} \) is less than \( \text{counter} \), then, it means that the server \( ip \) has not answered the request before the timeout and thus the component \( \text{req} \cdot \text{packet} \) component is concatenated with another element of the sequence, which states that there was no answer from the server, and the \( \text{counter} \) component is incremented by one unit. However, if the size of the sequence \( \text{req} \cdot \text{packet} \) is equal to \( \text{counter} \), it means that there has been an answer from \( ip \), and then, the \( \text{counter} \) component is incremented by one unit. (2) If the number of requests, \( \text{counter} \) has reached the maximal number of requests \( \text{max} \cdot \text{rep} \), then, a signal \( \text{Skip} \) ends the recursion and terminates the Circus action \( \text{Run} \).

\[
\text{Run} \equiv \mu X \cdot \left( \begin{array}{c}
\text{if}(\text{counter} < \text{max} \cdot \text{rep}) \rightarrow \\
\text{Send} \quad 1000 \\
\triangleright \\
\text{display} \cdot \text{resp} \cdot ! \cdot (\text{TIMEOUT}, \text{time}) \\
\text{counter} := \text{counter} + 1 \\
\parallel (\text{counter} = \text{max} \cdot \text{rep}) \rightarrow \text{Skip} \\
\text{fi}
\end{array} \right)
\]

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

\[
\bullet \quad \text{InitSt} ; \quad \text{Run}
\]

end
In this example we give an overview of a *Circus* process. During the construction of our 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 three 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
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 iseq1[X] is presented below.

$$iseq1[X] == \{ s : \text{seq} X \mid s \neq \langle \rangle \land s \in \mathbb{N} \rightarrow X \}$$

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 \texttt{DecOrHexValue}\texttt{Type} as the set of integers. The type \texttt{IdentifierValue}\texttt{Type}, a new name for the type \texttt{DecOrHexValue}\texttt{Type}, is the type of identifiers in the configuration table.

\[
\texttt{DecOrHexValue}\texttt{Type} == \mathbb{Z}
\]

We define the type \texttt{Name}\texttt{Type} for names of the partitions and ARINC 653 processes, which is type of \texttt{String}.

\[
\{\text{String}\}
\]

\texttt{Name}\texttt{Type} == \texttt{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 \texttt{SYSTEM\_TIME\_TYPE}, which is defined as \texttt{DecOrHexValue}\texttt{Type}: integer numbers with a minimum time interval of one nanosecond.

\[
\texttt{SYSTEM\_TIME\_TYPE} == \texttt{DecOrHexValue}\texttt{Type}
\]

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 \texttt{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.

\[
\texttt{RETURN\_CODE\_TYPE} ::= \texttt{NO\_ERROR} | \texttt{NO\_ACTION} | \texttt{NOT\_AVAILABLE}
| \texttt{INVALID\_PARAM} | \texttt{INVALID\_CONFIG}
| \texttt{INVALID\_MODE} | \texttt{TIMED\_OUT}
\]

There are some constants that are used in multiple types. For that, we create the type \texttt{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 \texttt{ARINC\_CONSTANTS}.

\[
\texttt{ARINC\_CONSTANTS} ::= \texttt{COLD\_START} | \texttt{WARM\_START} | \texttt{COLD\_RESTART}
| \texttt{WARM\_RESTART} | \texttt{IDLE} | \texttt{NORMAL}
| \texttt{ERROR\_MODE} | \texttt{IGNORE} | \texttt{SHUTDOWN}
| \texttt{RESET} | \texttt{ARINC\_CONSTANTS\_NULL}
\]

As an example, we define the possible operation modes of a partition. The possible operating modes are \texttt{NORMAL}, \texttt{IDLE}, \texttt{COLD\_START} and \texttt{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.

\[
\text{OPERATING\_MODE\_TYPE} == \text{ARINC\_CONSTANTS} \setminus \{\text{COLD\_RESTART, WARM\_RESTART, ERROR\_MODE, IGNORE, SHUTDOWN, RESET, ARINC\_CONSTANTS}\_NULL\}\n\]

\[
\text{OPERATING\_INIT\_MODE\_TYPE} == \text{ARINC\_CONSTANTS} \setminus \{\text{IDLE, NORMAL, COLD\_RESTART, WARM\_RESTART, ERROR\_MODE, IGNORE, SHUTDOWN, RESET, ARINC\_CONSTANTS}\_NULL\}\n\]

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.

\[
\text{START\_CONDITION\_TYPE ::= NORMAL\_START | PARTITION\_RESTART | HM\_NORMAL\_START | HM\_PARTITION\_RESTART}
\]

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 DecOrHexValue\_Type.

\[
\text{LOCK\_LEVEL\_TYPE == DecOrHexValue\_Type}
\]

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.

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.

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.

process A653_Partition $\equiv$ partitionId : PARTITION_ID_TYPE;
partHM : A653_PartitionHMTableType;
sysError : seq1 A653_ErrorIdentifierType $\bullet$ begin
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`.

\[
\text{GetTime} \triangleq \\
\quad \text{proc\_req\_system\_time}倒是\\\text{partitionId}\?\text{processId} \rightarrow \\
\quad \text{part\_req\_system\_time}倒是\\\text{partitionId} \rightarrow \\
\quad \text{part\_get\_system\_time}倒是\\\text{partitionId}\?\text{t} \rightarrow \\
\quad \text{proc\_get\_system\_time}倒是\\\text{partitionId}\?\text{processId}\!\text{t} \rightarrow \\
\quad \text{return\_code}倒是\\\text{partitionId}\!\text{rc} \rightarrow \text{Skip}\]

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.

\[
\text{SetPartitionMode} \triangleq \\
\quad \text{proc\_set\_partition\_mode}倒是\\\text{partitionId}\?\text{processId}\!\text{OPERATING_MODE} \rightarrow \\
\quad \text{part\_set\_partition\_mode}倒是\\\text{partitionId}\!\text{OPERATING_MODE} \rightarrow \\
\quad \text{return\_code}倒是\\\text{partitionId}\!\text{rc} \rightarrow \text{Skip}\]

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 \textit{APEX}.

\textit{GetPartitionStatus} \( \triangleq \)
\[
\begin{align*}
&\text{proc}\_\text{req}\_\text{partition}\_\text{status}\_\text{partitionId}\_?\_\text{processId} \rightarrow \\
&\text{part}\_\text{req}\_\text{partition}\_\text{status}\_\text{partitionId} \rightarrow \\
&\text{part}\_\text{get}\_\text{partition}\_\text{status}\_\text{partitionId}\_?\_\text{st} \rightarrow \\
&\text{proc}\_\text{get}\_\text{partition}\_\text{status}\_\text{partitionId}\_\text{processId}\_\text{st} \rightarrow \\
&\text{return}\_\text{code}\_\text{partitionId}\_?\_\text{rc} \rightarrow \text{Skip}
\end{align*}
\]

The \textit{Circus} action \textit{GetSamplingPortId} obtains the identifier of a sampling port of the partition in execution, through request to the \textit{APEX} service \textit{GET\_SAMPLING\_PORT\_ID}. The internal process in execution within a partition, \textit{processId}, requests the identifier associated to the sampling port name, \textit{SAMPLING\_PORT\_NAME}, through the channel \textit{proc}\_\text{req}\_\text{sampling}\_\text{port}\_\text{id}. Then, the channel \textit{part}\_\text{req}\_\text{sampling}\_\text{port}\_\text{id} sends the request to the \textit{APEX}. Finally, the channel \textit{part}\_\text{get}\_\text{sampling}\_\text{port}\_\text{id} receives the identifier, \textit{spid} from the \textit{APEX} and then sends it back to the process \textit{processId} through the channel \textit{proc}\_\text{get}\_\text{sampling}\_\text{port}\_\text{id}.

\textit{GetSamplingPortId} \( \triangleq \)
\[
\begin{align*}
&\text{proc}\_\text{req}\_\text{sampling}\_\text{port}\_\text{id}\_\text{partitionId}\_?\_\text{processId}\_?\_\text{SAMPLING\_PORT\_NAME} \rightarrow \\
&\text{part}\_\text{req}\_\text{sampling}\_\text{port}\_\text{id}\_\text{partitionId}\_?\_\text{SAMPLING\_PORT\_NAME} \rightarrow \\
&\text{part}\_\text{get}\_\text{sampling}\_\text{port}\_\text{id}\_\text{partitionId}\_?\_\text{spid} \rightarrow \\
&\text{proc}\_\text{get}\_\text{sampling}\_\text{port}\_\text{id}\_\text{partitionId}\_?\_\text{processId}\_?\_\text{spid} \rightarrow \\
&\text{return}\_\text{code}\_\text{partitionId}\_?\_\text{rc} \rightarrow \text{Skip}
\end{align*}
\]

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

\textit{ExecPartitionServices} \( \triangleq \)
\[
\begin{align*}
&\text{execPartition}\_\text{partitionId} \rightarrow \\
&\left( \mu X \cdot \left( \begin{array}{c}
\text{SetPartitionMode} \\
\square \text{GetPartitionStatus} \\
\square \text{GetSamplingPortId} \\
\square \text{GetTime} \\
\triangle \left( \text{interruptPartition}\_\text{partitionId} \rightarrow \text{Skip} \right)
\end{array} \right) ; X \right)
\end{align*}
\]

The \textit{Circus} action \textit{ExecPartition} describes the behaviour of an ARINC partition during the life time of a module. The action starts with a signal \textit{initPartition} sent by the operating system, with the identifier of the partition, \textit{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 \texttt{part\_send\_mpHM\_table}. Then, it recurses offering the \textit{Circus} action \texttt{ExecPartitionServices} in each iteration. It may be interrupted by either a signal \texttt{endPartition} indicating that the partition is being disabled or either \texttt{reinitPartition} followed by the action \texttt{ReinitPartition} indicating that the partition is being reinitialised.

\begin{equation}
\begin{aligned}
\text{ExecPartition} & \triangleq \text{initPartition.partitionId} \rightarrow \\
& \text{part\_send\_partHM\_table.partitionId!pHM!sysError} \rightarrow \\
& \text{moduleEndInit} \rightarrow \\
& (\mu X \bullet \text{ExecPartitionServices} ; X) \\
& \triangle (\text{endPartition.partitionId} \rightarrow \text{Skip} \\
& \square \text{reinitPartition.partitionId} \rightarrow \text{ReinitPartition})
\end{aligned}
\end{equation}

The \textit{Circus} action \texttt{ReinitPartition} starts with a signal \texttt{partHM\_restart\_partition} from the Health Monitor. Then, it recurses offering the \textit{Circus} action \texttt{ExecPartitionServices} in each iteration and may be interrupted by either a signal \texttt{endPartition} indicating that the partition is being disabled or \texttt{reinitPartition} followed by the action \texttt{ReinitPartition}.

\begin{equation}
\begin{aligned}
\text{ReinitPartition} & \triangleq \text{partHM\_restart\_partition.partitionId} \rightarrow \\
& (\mu X \bullet \text{ExecPartitionServices} ; X) \\
& \triangle (\text{endPartition.partitionId} \rightarrow \text{Skip} \\
& \square \text{reinitPartition.partitionId} \rightarrow \text{ReinitPartition})
\end{aligned}
\end{equation}

The main action of the \texttt{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 \texttt{moduleInit} and then, the \textit{Circus} action \texttt{ExecPartition} is executed until the interruption from the APEX through the \texttt{moduleEnd} channel indicating that the module will be switched off.

\begin{itemize}
  \item \texttt{moduleInit} \rightarrow (\texttt{ExecPartition} \triangle \texttt{moduleEnd} \rightarrow \texttt{Skip})
\end{itemize}

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 \texttt{PartitionHM Circus} processes, related to each partition running under the same module. The \textit{Circus} process \texttt{PartitionHM} is composed by a state \texttt{PartitionHMSSt} which stores the values of the system errors, \texttt{sysError}, and the possible recovery actions for that partition, \texttt{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 \texttt{PartitionHM Circus} process consists in an input channel, \texttt{part\_send\_partHM\_table} with the values of possible errors, \texttt{se}, and recovery actions, \texttt{phm}. Then the state variables are updated with these values. At any moments, the \texttt{PartitionHM} process may be interrupted by a signal \texttt{moduleEnd} or \texttt{endPartition}.
process PartitionHM ≜ partitionId : PARTITION_ID_TYPE • begin

state PartitionHMSt == [pHM : A653_PartitionHMTableType;
    sysError : seq₁ A653_ErrorIdentifierType]

• (part_send_partHM_table.partitionId? phm? se → )
  pHM, sysError := phm, se

△ (moduleEnd → Skip
  □ endPartition.partitionId → Skip )

end

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.

channelset PartitionHMChannels ==
  { part_send_partHM_table, moduleEnd, endPartition }

We define the layer that contains the ARINC partitions as the Partition_Layer Circus process. It has three inputs: the parameter partitionIds of type P 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.

process Partitions_Layer ≜ partitionIds : P PARTITION_ID_TYPE;
    partHM : seq₁ A653_PartitionHMTableType;
    sysError : seq₁ A653_ErrorIdentifierType

  ( || pid : partitionIds | { moduleInit, moduleEnd, moduleEndInit } |
  • ( A653_Partition(pid, partHM(pid), sysError )
    PartitionHMChannels | PartitionHM(pid ) )

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).

\[
\text{process APEX} \equiv \text{sysError : seq A653_ErrorIdentifierType;}
\text{mpHM : seq A653_MultiPartitionHMTTableType • begin}
\]

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.

\[
\text{GET\_TIME} \equiv \text{part\_req\_system\_time?pid} \rightarrow \\
\text{apex\_req\_system\_time.pid} \rightarrow \\
\text{apex\_get\_system\_time.pid?t} \rightarrow \\
\text{part\_get\_system\_time.pid?t} \rightarrow \\
\text{return\_code.pid!NO\_ERROR} \rightarrow \text{Skip}
\]

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 \( \text{nopm} \) 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 \( \text{nopm} \) 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 \( \text{opm} \) is NORMAL and the value of \( \text{nopm} \) 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 \( \text{nopm} \) 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 \( \text{nopm} \) 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 \( \text{nopm} \) is not NORMAL and the new value of \( \text{nopm} \) is NORMAL, then the channel apex\_req\_new\_partition\_mode sends the \( \text{nopm} \) value to the operating system and the return\_code channel sends a signal with a NO\_ERROR value.
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.

\[
\text{GET\_PARTITION\_STATUS} \equiv \\
\text{part\_req\_partition\_status?pid?nopm} \rightarrow \\
\text{apex\_req\_partition\_status.pid} \rightarrow \\
\text{apex\_get\_partition\_status.pid?opm} \rightarrow \\
\begin{cases} 
\text{if } (\text{nopm} \notin \text{OPERATING\_MODE\_TYPE}) \rightarrow \\
\text{return\_code.pid!INVALID\_PARAM} \rightarrow \text{Skip} \\
\text{(opm = \text{NORMAL} \land \text{nopm} = \text{NORMAL}) } \rightarrow \\
\text{apex\_new\_partition\_mode\_fail.pid} \rightarrow \\
\text{return\_code.pid!NO\_ACTION} \rightarrow \text{Skip} \\
\text{(opm = \text{COLD\_START} \land \text{nopm} = \text{WARM\_START}) } \rightarrow \\
\text{apex\_new\_partition\_mode\_fail.pid} \rightarrow \\
\text{return\_code.pid!INVALID\_MODE} \rightarrow \text{Skip} \\
\text{(nopm = IDLE} \\
\lor \text{nopm} \in \text{OPERATING\_INIT\_MODE\_TYPE} \rightarrow \\
\text{apex\_req\_new\_partition\_mode.pid!nopm} \rightarrow \\
\text{return\_code.pid!NO\_ERROR} \rightarrow \text{Skip} \\
\text{opm \neq \text{NORMAL} \land \text{nopm} = \text{NORMAL}) } \rightarrow \\
\text{apex\_req\_new\_partition\_mode.pid!nopm} \rightarrow \\
\text{return\_code.pid!NO\_ERROR} \rightarrow \text{Skip} \\
\end{cases}
\]

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...
3.3. THE APEX

axe_req_sampling_port_id. The operating system responds through the channel
axe_get_sampling_port_id with a number, spid, that is verified: if the received iden-
tifier 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.

\[
\text{GET\_SAMPLING\_PORT\_ID} \equiv \\
\text{part\_req\_sampling\_port\_id?pid?spn} \rightarrow \\
\text{axe_req\_sampling\_port\_id?spn} \rightarrow \\
\text{axe_get\_sampling\_port\_id?spid} \rightarrow \\
\begin{cases} 
\text{if (spid = -1)} & \rightarrow \\
\text{part\_get\_sampling\_port\_id\_pid\_spid} & \rightarrow \\
\text{return\_code\_pid\!INVALID\_CONFIG} & \rightarrow \text{Skip} \\
\text{[] (spid \neq -1)} & \rightarrow \\
\text{part\_get\_sampling\_port\_id\_pid\_spid} & \rightarrow \\
\text{return\_code\_pid\!NO\_ERROR} & \rightarrow \text{Skip} \\
\end{cases}
\]

The collection of APEX services are defined by the ExecApexServices action. The ac-
tion 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.

\[
\text{ExecApexServices} \equiv \text{execPartition?pid} \rightarrow \\
\text{\(\mu X \bullet \begin{cases} \text{GET\_TIME} \\ \Box \text{SET\_PARTITION\_MODE} \\ \Box \text{GET\_PARTITION\_STATUS} \\ \Box \text{GET\_SAMPLING\_PORT\_ID} \\
\Delta (\text{interruptPartition\_pid} \rightarrow \text{Skip}) \\
\Box \text{endPartition\_pid} \rightarrow \text{Skip}
\end{cases} ; X \) \)
\]

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 pro-
cess starts the ExecApex sending the set of errors and recovery actions to the Multi-
Partition health monitor, through the channel axe_send_mpHM_table, and then, a sig-
nal 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.

\[
\text{ExecApex} \equiv \text{axe_send_mpHM_table!mpHM!sysError} \rightarrow \\
\text{moduleEndInit} \rightarrow (\mu X \bullet \text{ExecApexServices} ; X)
\]
The APEX is initialised with a signal from the operating system, via the channel \texttt{moduleInit}, indicating that the module is being initialised. Then, the \texttt{ExecApex} is executed during the lifetime of the APEX Circus process until it receives a signal to terminate the process via the channel \texttt{moduleEnd}.

\begin{itemize}
  \item \texttt{moduleInit} $\rightarrow$ (\texttt{ExecApex} \ $\Delta$ \ \texttt{moduleEnd} $\rightarrow$ \texttt{Skip})
\end{itemize}

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

\begin{verbatim}
process MultiPartitionHM $\triangleq$ begin

  state MultiPartitionHMSt ==
    [mpHM : seq\_1 A653\_MultiPartitionHMTableType;
     sysError : seq\_1 A653\_ErrorIdentifierType]

  $\bullet$ apex\_send\_mpHM\_table?mp?se $\rightarrow$
    mpHM, sysError := mp, se
    $\Delta$moduleEnd $\rightarrow$ \texttt{Skip}

end
\end{verbatim}

We define below the Circus channel set \texttt{MultiPartitionHMChannels} for communication between the APEX and the MultiPartitionHM Circus process.

\begin{verbatim}
channelset MultiPartitionHMChannels == \{
  apex\_send\_mpHM\_table, moduleEnd \}
\end{verbatim}

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{verbatim}
process APEX\_Layer $\triangleq$ mpHM : seq\_1 A653\_MultiPartitionHMTableType;
    sysError : seq\_1 A653\_ErrorIdentifierType
  $\bullet$
    \left( \begin{array}{c}
      \text{APEX}(\text{sysError}, \text{mpHM}) \\
      \text{\{MultiPartitionHMChannels \} \text{MultiPartitionHM}}
    \end{array} \right)
\end{verbatim}

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.

```
A653_PartitionBaseType

  Identifier : PARTITION_ID_TYPE
  Name : NameType
```

A Partition is composed of four components: a PartitionDefinition of type A653_PartitionBaseType, 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[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.

```
Partition

  PartitionDefinition : A653_PartitionBaseType
  PartitionPeriodicity : A653_PartitionPeriodicityType
  MemoryRegions : iseq[A653_MemoryRegionType]
  PartitionPorts : iseq[PartitionPort]
```

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 \textit{PartitionsType} is specified a non-empty injective sequence of partitions (iseq\([\text{Partition}])$: there must be at least one partition allocated in that ARINC 653 module.

\textit{PartitionsType} \textbf{==} iseq\([\text{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 \textit{Name} of the partition, \textit{Duration} of execution, the \textit{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 \textit{A653\_PartitionTimeWindowType} schema below.

\begin{verbatim}
A653_ParticitionTimeWindowType
PartitionNameRef : NameType;
Duration, Offset : DecOrHexValueType;
PeriodicProcessingStart : Boolean
\end{verbatim}

The schedule for execution of the partitions within an ARINC 653 module, \textit{ScheduleType}, is defined as a non-empty injective sequence of \textit{A653\_PartitionTimeWindowType}.

\textit{ScheduleType} \textbf{==} iseq\([\text{A653\_PartitionTimeWindowType}])

The \textit{HealthMonitoringType} component contains records of the set of possible errors, \textit{SystemErrors}, and recovery actions for these errors. The recovery actions are split in three levels: \textit{ModuleHM} contains records of the possible recovery actions for errors that affects the module itself; \textit{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 \textit{PartitionHM} stores the recovery actions to be applied only within the boundaries of the partition in execution.

\begin{verbatim}
HealthMonitoringType
SystemErrors : iseq\([\text{A653\_ErrorIdentifierType}])
ModuleHM : iseq\([\text{A653\_ModuleHMTableType}])
MultiPartitionHM : iseq\([\text{A653\_MultiPartitionHMTableType}])
PartitionHM : iseq\([\text{A653\_PartitionHMTableType}])
\end{verbatim}
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.

<table>
<thead>
<tr>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name : NameType</td>
</tr>
<tr>
<td>Partitions : PartitionsType</td>
</tr>
<tr>
<td>Schedules : ScheduleType</td>
</tr>
<tr>
<td>HealthMonitoring : HealthMonitoringType</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>PartitionVariables</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPERATING_MODE : OPERATING_MODE_TYPE</td>
</tr>
<tr>
<td>START_CONDITION : START_CONDITION_TYPE</td>
</tr>
<tr>
<td>LOCK_LEVEL : LOCK_LEVEL_TYPE</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>PARTITION_STATUS_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier : PARTITION_ID_TYPE</td>
</tr>
<tr>
<td>Period : SYSTEM_TIME_TYPE</td>
</tr>
<tr>
<td>Duration : SYSTEM_TIME_TYPE</td>
</tr>
<tr>
<td>PartitionVariables</td>
</tr>
</tbody>
</table>

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 \textit{OperatingSystem}. The process has a single input, \textit{module} of type \textit{Module}, presented above.

\begin{center}
\begin{tabular}{c}
\textbf{process} \textit{OperatingSystem} \equiv \textit{module} : \textit{Module} \bullet \textbf{begin} \\
\end{tabular}
\end{center}

The state of the \textit{OperatingSystem} Circus process comprises the following components: a non-empty injective sequence of \textit{PartitionVariables}; the \textit{major\_time\_frame} of the execution of the partitions; the current time in the system, \textit{system\_time}; the \textit{current\_partition} in execution within the module according to the \textit{Schedules}; the identifier of the partition in execution, \textit{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 \textit{partitions\_variables} is equal to the number of \textit{Partitions} allowed to execute within the module.

\begin{center}
\begin{tabular}{c}
\textit{OSSt} \\
partitions\_variables : \textit{iseq}[\textit{PartitionVariables}] \\
major\_time\_frame : \textit{SYSTEM\_TIME\_TYPE} \\
system\_time : \textit{SYSTEM\_TIME\_TYPE} \\
current\_partition : \textit{A653\_Partition\_Time\_Window\_Type} \\
current\_partition\_id : \textit{PARTITION\_ID\_TYPE} \\
sampling\_ports : (\textit{SAMPLING\_PORT\_ID\_TYPE} \mapsto \textit{SAMPLING\_PORT\_NAME\_TYPE}) \\
queuing\_ports : (\textit{QUEUING\_PORT\_ID\_TYPE} \mapsto \textit{QUEUING\_PORT\_NAME\_TYPE}) \\
\# partitions\_variables = \# \textit{module}.Partitions
\end{tabular}
\end{center}

\textbf{state} \textit{OSSt}

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

\begin{center}
\textit{InitPartition} \equiv \textbf{|||} \textit{pid} : (1 \ldots \textit{\# module}.Partitions) \bullet \textit{initPartition}.\textit{pid} \rightarrow \textbf{Skip}
\end{center}

In order to calculate the major time frame, we define the Circus action \textit{MajorTimeFrame} modelled as a Z schema. We first declare that the state is being updated (\textit{\Delta OSSt}) and we declare the variable \textit{getMajorTimeFrame} of type \textit{PartitionsType}. Then we introduce three predicates: firstly, the variable \textit{getMajorTimeFrame} contains the same elements of \textit{module}.Partitions; secondly, \textit{getMajorTimeFrame} is ordered by the \textit{Period} in
3.4. THE OPERATING SYSTEM

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 θ(OSSt \ {major_time_frame})' = θ(OSSt \ {major_time_frame}) denotes that besides major_time_frame, the rest of the state variables remain unchanged after the execution of MajorTimeFrame.

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.

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 ≡ updateClock?x → 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*.

\[
\text{OSGetTime} \equiv \text{apex_req_system_time.current_partition_id} \rightarrow \text{UpdateSystemTime}; \quad \text{apex_get_system_time.current_partition_id!system_time} \rightarrow \text{Skip}
\]

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 (\( \Xi \text{OSSt} \)). 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.

\[
\begin{align*}
\text{PartitionStatus} & \quad \Xi \text{OSSt} \\
\var p & \colon \text{PARTITION_STATUS_TYPE} \\
(p!).\text{Identifier} &= \quad (\text{module.Partitions(current_partition_id)}).\text{PartitionDefinition.Identifier} \\
(p!).\text{Period} &= \quad (\text{module.Partitions(current_partition_id)}).\text{PartitionPeriodicity.Period} \\
(p!).\text{Duration} &= \quad (\text{module.Partitions(current_partition_id)}).\text{PartitionPeriodicity.Duration} \\
(p!).\text{LOCK_LEVEL} &= \quad (\text{partitions_variables(current_partition_id)}).\text{LOCK_LEVEL} \\
(p!).\text{OPERATING_MODE} &= \quad (\text{partitions_variables(current_partition_id)}).\text{OPERATING_MODE} \\
(p!).\text{START_CONDITION} &= \quad (\text{partitions_variables(current_partition_id)}).\text{START_CONDITION}
\end{align*}
\]

\[
\text{OSGetPartitionStatus} \equiv \text{var } p \colon \text{PARTITION_STATUS_TYPE} \bullet \\
\text{apex_req_partition_status.current_partition_id} \rightarrow (\text{PartitionStatus}); \\
\text{apex_get_partition_status.current_partition_id!p} \rightarrow \text{Skip}
\]

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*

\[
\text{SetPMode} \equiv \text{var } p \colon \text{PARTITION_STATUS_TYPE} \bullet \\
\text{apex_set_partition_mode.current_partition_id} \rightarrow (\text{PartitionStatus}); \\
\text{apex_get_partition_mode.current_partition_id!p} \rightarrow \text{Skip}
\]
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 `⊕` 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`.

```
\[ \Delta \text{OSSt} \]
\[ \text{nopm?} : \text{OPERATING_MODE_TYPE} \]
\[ \text{partitions_variables}' = \]
\[ \text{(let } o = \{ \text{OPERATING_MODE} == \text{nopm?}, \]
\[ \text{START_CONDITION} = \]
\[ \text{(partitions_variables(current_partition_id)).START_CONDITION}, \]
\[ \text{LOCK_LEVEL} = \]
\[ \text{(partitions_variables(current_partition_id)).LOCK_LEVEL} \]
\[ \bullet \text{partitions_variables ⊕ \{current_partition_id } \mapsto o} \} \]
\[ \theta(\text{OSSt} \setminus (\text{current_partition_id}, \text{current_partition, partitions_variables}))' = \]
\[ \theta(\text{OSSt} \setminus (\text{current_partition_id}, \text{current_partition, partitions_variables})) \]
```

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`.

```
\text{OSSetPartitionMode} \triangleq \text{var} \text{opm} : \text{OPERATING_MODE_TYPE} \bullet
\text{apex_req_partition_mode.current_partition_id} \rightarrow
\text{apex_get_partition_mode.current_partition_id}!(\)
\text{PartitionsVariables(current_partition_id)}, \text{OPERATING_MODE} \rightarrow
\begin{cases}
\text{if}(\text{nopm} = \text{IDLE}) \rightarrow \\
\text{endPartition.current_partition_id} \rightarrow (\text{SetPMode})
\end{cases}
\begin{cases}
\text{if}(\text{nopm} \in \text{OPERATING_INIT_MODE_TYPE}) \rightarrow \\
\text{reinitPartition.current_partition_id} \rightarrow (\text{SetPMode})
\end{cases}
\begin{cases}
\text{if}(\text{nopm} = \text{NORMAL}) \rightarrow (\text{SetPMode})
\end{cases}
\text{fi}
\begin{cases}
\text{if}(\text{apex_new_partition_mode.fail?current_partition_id} \rightarrow \text{Skip})
\end{cases}
```

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.

```
OSGetSamplingPortId ≜
apex_req_sampling_port_id.current_partition_id?spn →
(if (spn ∈ ran sampling_ports) →
apex_get_sampling_port_id.current_partition_id!(μ
x : (dom(sampling_ports ⊲ {spn})) • x) →
return_code.current_partition_id!NO_ERROR → Skip
)\( (spn ∉ ran sampling_ports) →
apex_get_sampling_port_id.current_partition_id!(-1) →
return_code.current_partition_id!INVALID_CONFIG → Skip
\)
```

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`.

```
ExecOSServices ≜
execPartition.current_partition_id →
(μ X • (OSGetTime
\□ OSGetPartitionStatus
\□ OSSetPartitionMode
\□ OSGetSamplingPortId) ; X )
```

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 $\text{MajorTimeFrame}$, then it recurses as follows: at each iteration, the action $\text{UpdateSystemTime}$ updates the current time within the module, then $\text{NextPartition}$ 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 $\text{ExecOS}$ 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 $\text{interruptPartition}$ for that partition in execution and then the action recurses.

$$\text{ExecOS} \triangleq \text{var getMajorTimeFrame : PartitionsType} \bullet (\text{MajorTimeFrame}) ;\begin{align*}
\mu X \bullet & \text{UpdateSystemTime} ; (\text{NextPartition}) ; \left( \begin{array}{c}
\text{if} \left( \text{current\_partition.Off}\text{set} > \left( \text{system\_time}\ mod \text{major\_time\_frame} \right) \right) \rightarrow \\
\text{wait} \left( \text{current\_partition.Off}\text{set} - \left( \text{system\_time}\ mod \text{major\_time\_frame} \right) \right) ; X \\
\text{fi} \\
\end{array} \right)
\end{align*}$$

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

$$\bullet \text{moduleInit} \rightarrow \text{os\_send\_modHM\_table}!(\text{module\.HealthMonitoring\.ModuleHM})(\text{module\.HealthMonitoring\.SystemErrors}) \rightarrow (\text{ExecOS} \triangle \text{moduleEnd} \rightarrow \text{Skip})$$

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 $\text{Timer}$.

$$\text{process } \text{Timer} \triangleq \begin{align*}
\end{align*}$$
We define that the Timer state, \( \text{TimerSt} \), is composed by a single component, \( \text{clock} \) that contains records of the current system time.

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

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

\[
\text{Counter} \triangleq \left( \mu X \bullet \left( \text{wait}(1) ; \text{clock} := \text{clock} + 1 \right) \parallel \left( \mu Y \bullet \text{updateClock}! \text{clock} \rightarrow Y \uparrow \text{Skip} \right) \right) ; X
\]

The Timer process starts with a signal \( \text{moduleInit} \), then it executes \( \text{Counter} \) until the interruption by a signal \( \text{moduleEnd} \).

\[
\bullet \text{moduleInit} \rightarrow (\text{Counter} \triangle \text{moduleEnd} \rightarrow \text{Skip})
\]

\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 \textit{MultiPartitionHM Circus} process, we model the \textit{ModuleHM Circus} process that recovers the faults and failures within the module level. At the current stage of our work, the \textit{ModuleHM Circus} process only receives the list of errors and recovery actions from the \textit{OperatingSystem Circus} process through the channel \( os\_send\_mHM\_table \) and then assigns these values to the state variables \( mHM \) and to \( sysError \). This \textit{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.

\[
\text{process } \text{ModuleHM} \triangleq \begin{align*}
\text{state } \text{ModuleHMSt} == & \left[ \text{modHM} : \text{seq}_1 \textit{A653\_ModuleHMTableType}; \\
& \text{sysError} : \text{seq}_1 \textit{A653\_ErrorIdentifierType} \right] \\
\bullet & \left( os\_send\_modHM\_table?mp?se \rightarrow \text{modHM} , \text{sysError} := mp, se \right) \\
& \triangle \text{moduleEnd} \rightarrow \text{Skip}
\end{align*}
\]

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

\[
\text{channelset ModuleHMChannels} == \{ \text{os\_send\_modHM\_table, moduleEnd} \}
\]

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.

\[
\text{process OS\_Layer} \equiv \text{module : Module}
\]

\[
\begin{align*}
& (\text{OperatingSystem(module)} \\
& \quad [\{\text{updateClock}\}] ) \\
& \quad \text{Timer} \\
& \quad [\text{ModuleHMChannels } \text{ModuleHM}]
\end{align*}
\]

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.

\[
\text{process IMA\_Module} \equiv \text{module : Module}
\]

\[
\begin{align*}
& (\text{OS\_Layer(module)} \\
& \quad [\text{ApexOS}] \\
& \quad \text{APEX\_Layer} (\text{module\_HealthMonitoring\_MultiPartitionHM, module\_HealthMonitoring\_SystemErrors}) \\
& \quad [\text{PartitionApex}] \\
& \quad \text{Partitions\_Layer} (1..\#\text{module\_Partitions, module\_HealthMonitoring\_PartitionHM, module\_HealthMonitoring\_SystemErrors})
\end{align*}
\]

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 \( \text{DecOrHexValueType} \) as a subset of natural numbers, ranging from 0 up to 10.

\[
\text{DecOrHexValueType} = \{0..10\}
\]

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

\[
\text{datatype StringAccessRights} = \text{READONLY} | \text{READ_WRITE} | \text{NoAccessRights}
\]

Then we model the type \( \text{String} \), illustrated below, as free types such as the presented above \( \text{StringAccessRights} \).

\[
\text{datatype String} = \text{SNT.StringNameType} | \text{SAR.StringAccessRights} \\
\text{ | SMRT.StringMemoryRegionType} | \text{SMRN.StringMemoryRegionName} \\
\text{ | SPNT.StringSAMPLING_PORT_NAME_TYPE} | \text{QPNT.StringQUEUING_PORT_NAME_TYPE}
\]

The translation of the free type \( \text{ARINC_CONSTANTS} \) is pretty straightforward from our Circus model.

\[
\text{datatype ARINC_CONSTANTS} = \text{COLD_START} | \text{WARM_START} | \text{COLD_RESTART} | \text{WARM_RESTART} \\
\text{ | IDLE} | \text{NORMAL} | \text{ERROR_MODE} | \text{IGNORE} | \text{SHUTDOWN} | \text{RESET} \\
\text{ | ARINC_CONSTANTS_NULL}
\]

The translation of the subsets \( \text{OPERATING_MODE_TYPE} \) and \( \text{OPERATING_INIT_MODE_TYPE} \) is made using the operation \( \text{diff} \) which results in the subtraction of elements in curly brackets, \( \{\text{COLD_RESTART}, \text{WARM_RESTART}, ...\} \) from the set of \( \text{ARINC_CONSTANTS} \).

\[
\text{OPERATING_MODE_TYPE} = \text{diff(ARINC_CONSTANTS,} \\
\text{ \{COLD_RESTART, WARM_RESTART, ERROR_MODE,}  \\
\text{ IGNORE, SHUTDOWN, RESET, ARINC_CONSTANTS_NULL \}})
\]

\[
\text{OPERATING_INIT_MODE_TYPE} = \text{diff(ARINC_CONSTANTS,} \\
\text{ \{COLD_RESTART, WARM_RESTART, IDLE, NORMAL, ERROR_MODE,}  \\
\text{ IGNORE, SHUTDOWN, RESET, ARINC_CONSTANTS_NULL\}})
\]
3.5. TRANSLATING THE CIRCUS MODEL INTO CSP

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.

\[
\text{nametype Partition} = (\text{A653_PartitionBaseType}, \text{A653_PartitionPeriodicityType}, \text{MemoryRegion}, \text{PartitionPort})
\]

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, Partition1, Partition2, and Partition3 of type Partition.

\[
\begin{align*}
\text{Partitions1} &= ((\text{SPtN.PartitionName1}, 1), (2, 1), \text{PartitionPort1}, \text{MemoryRegionType1}) \\
\text{Partitions2} &= ((\text{SPtN.PartitionName2}, 2), (1, 1), \text{PartitionPort2}, \text{MemoryRegionType1}) \\
\text{Partitions3} &= ((\text{SPtN.PartitionName3}, 3), (3, 1), \text{PartitionPort3}, \text{MemoryRegionType1})
\end{align*}
\]

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 PartitionsType, which is a set of three sequences of partitions.

\[
\text{nametype PartitionsType} = \{<\text{Partitions1}>, <\text{Partitions1, Partitions2}>, <\text{Partitions1, Partitions2, Partitions3}>\}
\]

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.

\[
\text{nametype Module} = (\text{NameType}, \text{PartitionsType}, \text{ScheduleType}, \text{HealthMonitoringType})
\]

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).

\[
\begin{align*}
\text{getModuleName}(\text{Name, Partitions, Schedules, HealthMonitoring}) &= \text{Name} \\
\text{getPartitions}(\text{Name, Partitions, Schedules, HealthMonitoring}) &= \text{Partitions} \\
\text{getPartSchedules}(\text{Name, Partitions, Schedules, HealthMonitoring}) &= \text{Schedules} \\
\text{getHealthMonitoring}(\text{Name, Partitions, Schedules, HealthMonitoring}) &= \text{HealthMonitoring}
\end{align*}
\]

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`.

```csp
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.

```csp
PartitionHMStChannels =
```
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.

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.

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.

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.

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.

$$Update\_System\_Time = updateClock?x \rightarrow set\_system\_time!x \rightarrow SKIP$$

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 $Operating\_System\_Circus$ 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 $OSGet\_Time$ 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 $Update\_System\_Time$ 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 $apex\_get\_system\_time$.

$$OSGet\_Time = get\_current\_partition\_id?current\_partition\_id \rightarrow$$
$$\text{(apex\_req\_system\_time.current\_partition\_id \rightarrow Update\_System\_Time;}$$
$$\text{get\_system\_time?st \rightarrow apex\_get\_system\_time.current\_partition\_id!st \rightarrow SKIP})$$

The translation of the action $OSGet\_Partition\_Status$ does not use the auxiliary $Circus$ action $Partition\_Status$. 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 $Partitions\_Variables$, $pv$, 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 $Partitions\_Variables$ of the partition $current\_partition\_id$ with the values of that partition. We are able to construct the tuple of type $Partitions\_Variables$ by using the auxiliary operations such as $get\_Pid$ and $get\_Period$, which returns these values from the input tuple.

$$OSGet\_Partition\_Status =$$
$$get\_current\_partition\_id?current\_partition\_id \rightarrow$$
$$\text{(apex\_req\_partition\_status.current\_partition\_id \rightarrow)}$$
$$\text{get\_Partitions\_Variables?pv \rightarrow)}$$
$$\text{if (#pv > 0)}$$
$$\text{then apex\_get\_partition\_status.current\_partition\_id!}\{$$
$$\text{get\_Pid(get\_PartitionBase{}}$$
$$\text{get\_Partition(get\_Partitions(modl), current\_partition\_id))},$$
$$\text{get\_Period(get\_Partition\_Periodicity{}}$$
$$\text{get\_Partition(get\_Partitions(modl), current\_partition\_id))},$$
$$\text{get\_Duration(get\_Partition\_Periodicity{}}$$
$$\text{get\_Partition(get\_Partitions(modl), current\_partition\_id))},$$
$$\text{(get\_PV\_Oper\_Mode(get\_PS(pv, current\_partition\_id))},$$
$$\text{), SKIP})$$
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 allow 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.

\[
\text{Wait}(n) = \begin{cases} 
\text{tock} & \text{if } n > 0 \\
\text{SKIP} & \text{else}
\end{cases}
\]

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 recursively decrements 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 \(A \triangle timeout \rightarrow B\) and the Counter process, synchronising on timeout and tock events. The process \(A \triangle timeout \rightarrow B\) 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 (\(\backslash\)) with a signal timeout, triggered by Counter(t) followed by the process B after t elapses.

\[
\text{Timeout}(A,B,t) = 
\begin{cases} 
\text{let Counter}(t) = 
\begin{cases} 
\text{tock} & \text{if } (t>0) \\
\text{SKIP} & \text{else}
\end{cases} \\
\text{within } ((A \triangle timeout \rightarrow B) \backslash (\text{timeout, tock}) \backslash \text{Counter}(t)) \backslash \text{timeout}
\end{cases}
\]

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
3.5. TRANSLATING THE CIRCUS MODEL INTO CSP

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.

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

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.

\[
\text{OSMainAction = } \\
\text{moduleInit -> } \\
\text{cs_send_modHM_table!(getModuleHMTable(getHealthMonitoring(modl)))!!( } \\
\text{getSystemError(getHealthMonitoring(modl))) } \\
\text{-> (ExecOS /\ moduleEnd -> SKIP)} \\
\]

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.

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

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.

\[
\text{Module}(\text{modl}) = \text{OS_Layer } | \text{ApexOs} \]

\[
\text{Partitions_Layer}(\{|1..\#\text{getPartitions}(\text{modl})|, \text{getPartitionHMTable}(\text{getHealthMonitoring}(\text{modl})), \text{getSystemError}(\text{getHealthMonitoring}(\text{modl})) \ | \text{PartitionApex} \}
\text{APEX_Layer}(\text{getSystemError}(\text{getHealthMonitoring}(\text{modl})), \text{getMultiPartitionHMTable}(\text{getHealthMonitoring}(\text{modl})))))
\]

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 *tick* 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 there 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 analyze 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 au-
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 ‘×’ (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_Particle 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

\[ iseq[X] \equiv \{ s : \text{seq } X | s \neq \langle \rangle \land s \in \mathbb{N} \Rightarrow X \} \]

A.1.1 Basic types definition

1. Definition for numbers

\[ \text{DecOrHexValueType} \equiv \mathbb{Z} \]

2. Definition for Strings

\[
\begin{align*}
[\text{String}] \\
\text{NameType} \equiv \text{String}
\end{align*}
\]

3. Definition of Boolean

\[ \text{Boolean} \equiv \text{TRUE} | \text{FALSE} \]

4. Definition of Time

\[ \text{SYSTEM\_TIME\_TYPE} \equiv \text{DecOrHexValueType} \]

5. Definition of Partitions and Process Identifiers

\[
\begin{align*}
\text{PARTITION\_ID\_TYPE} & \equiv \text{DecOrHexValueType} \\
\text{PROCESS\_ID\_TYPE} & \equiv \text{DecOrHexValueType}
\end{align*}
\]
6. Definition of messages provided in the end of the execution of an APEX service

\[
\text{RETURN\_CODE\_TYPE ::= NO\_ERROR | NO\_ACTION | NOT\_AVAILABLE} \\
\quad | \text{INVALID\_PARAM | INVALID\_CONFIG} \\
\quad | \text{INVALID\_MODE | TIMED\_OUT}
\]

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

\[
\text{ARINC\_CONSTANTS ::= COLD\_START | WARM\_START | COLD\_RESTART} \\
\quad | WARM\_RESTART | IDLE | NORMAL \\
\quad | ERROR\_MODE | IGNORE | SHUTDOWN \\
\quad | RESET | ARINC\_CONSTANTS\_NULL}
\]

8. Definition of Operating modes, subsets of \text{ARINC\_CONSTANTS}:

\[
\text{OPERATING\_MODE\_TYPE == ARINC\_CONSTANTS \setminus \{COLD\_RESTART, WARM\_RESTART,} \\
\quad \text{ERROR\_MODE, IGNORE, SHUTDOWN,} \\
\quad \text{RESET, ARINC\_CONSTANTS\_NULL\}}
\]

\[
\text{OPERATING\_INIT\_MODE\_TYPE == ARINC\_CONSTANTS \setminus \{IDLE, NORMAL,} \\
\quad \text{COLD\_RESTART, WARM\_RESTART,} \\
\quad \text{ERROR\_MODE, IGNORE, SHUTDOWN,} \\
\quad \text{RESET, ARINC\_CONSTANTS\_NULL\}}
\]

9. Start Condition

\[
\text{START\_CONDITION\_TYPE ::= NORMAL\_START | PARTITION\_RESTART} \\
\quad | HM\_NORMAL\_START | HM\_PARTITION\_RESTART}
\]

10. Lock Level

\[
\text{LOCK\_LEVEL\_TYPE == DecOrHexValueType}
\]
11. Errors and levels of error definition

\[
\text{ErrorCodeType ::= DEADLINE\_MISSED | APPLICATION\_ERROR}
| NUMERIC\_ERROR | ILLEGAL\_REQUEST
| STACK\_OVERFLOW | MEMORY\_VIOLATION
| HARDWARE\_FAULT | POWER\_FAIL
\]

\[
\text{MODULE\_LEVELS ::= PARTITION | PROCESS}
| MODULE | MODULE\_LEVELS\_NULL
\]

PartitionHMTableErrorLevelType ==

\[
\text{MODULE\_LEVELS \{ MODULE\}
\]

MultiPartitionHMTableErrorLevelType ==

\[
\text{MODULE\_LEVELS \{ PARTITION\}
\]

12. Recovery Actions definition for Module and Partition levels

\[
\text{ModuleLevelErrorRecoveryActionType ==
ARINC\_CONSTANTS \{ COLD\_RESTART, WARM\_RESTART, COLD\_START, WARM\_START, IDLE, NORMAL, ERROR\_MODE, ARINC\_CONSTANTS\_NULL\}
\]

\[
\text{PartitionLevelErrorRecoveryActionType ==
ARINC\_CONSTANTS \{ COLD\_START, WARM\_START, NORMAL, ERROR\_MODE, SHUTDOWN, RESET, ARINC\_CONSTANTS\_NULL\}
\]

13. Port message types definition

\[
[\text{MESSAGE\_ADDR\_TYPE}]
SAMPLING\_PORT\_NAME\_TYPE == NameType
MESSAGE\_SIZE\_TYPE == DecOrHexValue\_Type
SAMPLING\_PORT\_ID\_TYPE == DecOrHexValue\_Type
VALIDITY\_TYPE ::= INVALID | VALID
QUEUING\_PORT\_ID\_TYPE == DecOrHexValue\_Type
[\text{QUEUING\_PORT\_NAME\_TYPE}]
QUEUING\_DISCIPLINE\_TYPE ::= FIFO | PRIORITY
MESSAGE\_RANGE\_TYPE == DecOrHexValue\_Type
WAITING\_RANGE\_TYPE == DecOrHexValue\_Type
PortDirection\_Type ::= SOURCE | DESTINATION
\]

A.1.2 Configuration Tables Data Structure

1. Partitions
   (a) Partition Status Type Definition
APPENDIX A. A CIRCUS MODEL OF THE ARINC 653 COMPONENTS

<table>
<thead>
<tr>
<th>A653_PartitionBaseType</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier : PARTITION_ID_TYPE</td>
</tr>
<tr>
<td>Name : NameType</td>
</tr>
</tbody>
</table>

(b) Partition Memory Region Type Definition

<table>
<thead>
<tr>
<th>A653_MemoryRegionType</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name, Type, AccessRights : NameType</td>
</tr>
<tr>
<td>Size : DecOrHexValueType</td>
</tr>
</tbody>
</table>

MemoryRegion == iseq[A653_MemoryRegionType]

(c) Partition Periodicity Type Definition

<table>
<thead>
<tr>
<th>A653_PartitionPeriodicityType</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period : SYSTEM_TIME_TYPE</td>
</tr>
<tr>
<td>Duration : SYSTEM_TIME_TYPE</td>
</tr>
</tbody>
</table>

(d) Partition Ports Types Definition

i. Port Base (used in both port types)

<table>
<thead>
<tr>
<th>PortBaseType</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name : NameType</td>
</tr>
<tr>
<td>MaxMessageSize : DecOrHexValueType</td>
</tr>
<tr>
<td>Direction : PortDirectionType</td>
</tr>
</tbody>
</table>

ii. Sampling Port

<table>
<thead>
<tr>
<th>A653_SamplingPortType</th>
</tr>
</thead>
<tbody>
<tr>
<td>PortBaseType</td>
</tr>
</tbody>
</table>

iii. Queuing Port

<table>
<thead>
<tr>
<th>A653_QueueingPortType</th>
</tr>
</thead>
<tbody>
<tr>
<td>PortBaseType</td>
</tr>
<tr>
<td>MaxNbMessage : DecOrHexValueType</td>
</tr>
</tbody>
</table>
iv. Partition Ports Definition

\[
\text{PartitionPort ::= SamplingPort}\langle \text{A653\_SamplingPortType} \rangle \\
| QueueingPort\langle \text{A653\_QueueingPortType} \rangle
\]

\[
\text{PartitionPorts} == \text{iseq}\{\text{PartitionPort}\}
\]

(e) Partition definition

\[
\text{Partition} \\
\{ \\
\text{PartitionDefinition : A653\_PartitionBaseType} \\
\text{PartitionPeriodicity : A653\_PartitionPeriodicityType} \\
\text{MemoryRegions : iseq}\{\text{A653\_MemoryRegionType}\} \\
\text{PartitionPorts : iseq}\{\text{PartitionPort}\}
\}
\]

(f) Sequence of Partitions definition

\[
\text{PartitionsType} == \text{iseq}\{\text{Partition}\}
\]

2. Partition Schedule

\[
\text{A653\_PartitionTimeWindowType} \\
\{ \\
\text{PartitionNameRef : NameType;} \\
\text{Duration, Offset : DecOrHexValueType;} \\
\text{PeriodicProcessingStart : Boolean}
\}
\]

\[
\text{ScheduleType} == \text{iseq}\{\text{A653\_PartitionTimeWindowType}\}
\]

3. Health Monitor

(a) Error Identifier Types

\[
\text{A653\_ErrorIdentifierType} \\
\{ \\
\text{ErrorIdentifier : IdentifierValueType} \\
\text{Description : NameType}
\}
\]

(b) Partition Health Monitor Definition
APPENDIX A. A CIRCUS MODEL OF THE ARINC 653 COMPONENTS

(c) Multi-Partition Health Monitor Definition

A653_PartitionHMTableType

| TableName : NameType |
| MultiPartitionHMTableNameRef : NameType |
| ErrorAction : seq1 PErrorActionType |

(d) Module Health monitor definition

A653_ModuleHMTableType

| StateIdentifier : IdentifierValueType |
| Description : NameType |
| ErrorAction : seq1 MErrorActionType |

(e) Health monitor Definition
A.2 CIRCUS CHANNELS

4. Module Definition

\[
\text{Module} \\
\quad \text{Name} : \text{NameType} \\
\quad \text{Partitions} : \text{PartitionsType} \\
\quad \text{Schedules} : \text{ScheduleType} \\
\quad \text{HealthMonitoring} : \text{HealthMonitoringType}
\]

5. Partition Status Type Definition

\[
\text{PartitionVariables} \\
\quad \text{OPERATING\_MODE} : \text{OPERATING\_MODE\_TYPE} \\
\quad \text{START\_CONDITION} : \text{START\_CONDITION\_TYPE} \\
\quad \text{LOCK\_LEVEL} : \text{LOCK\_LEVEL\_TYPE}
\]

\[
\text{PARTITION\_STATUS\_TYPE} \\
\quad \text{Identifier} : \text{PARTITION\_ID\_TYPE} \\
\quad \text{Period} : \text{SYSTEM\_TIME\_TYPE} \\
\quad \text{Duration} : \text{SYSTEM\_TIME\_TYPE} \\
\quad \text{PartitionVariables}
\]

A.2 Circus Channels

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

\[
\text{channel} \quad \text{moduleInit, moduleEndInit, moduleEnd}
\]

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

\[
\text{channel} \quad \text{initPartition, endPartition, reinitPartition} : \text{PARTITION\_ID\_TYPE} \\
\text{channel} \quad \text{execPartition, interruptPartition} : \text{PARTITION\_ID\_TYPE} \\
\text{channel} \quad \text{return\_code} : \text{PARTITION\_ID\_TYPE} \times \text{RETURN\_CODE\_TYPE}
\]
APPENDIX A. A **CIRCUS** MODEL OF THE ARINC 653 COMPONENTS

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

\[
\begin{align*}
\text{channel } & \text{apex}_\text{req}_\text{sampling}_\text{port}_\text{id} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
& \times \text{SAMPLING}\_\text{PORT}\_\text{NAME}\_\text{TYPE} \\
\text{channel } & \text{apex}_\text{get}_\text{sampling}_\text{port}_\text{id} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
& \times \text{SAMPLING}\_\text{PORT}\_\text{ID}\_\text{TYPE} \\
\text{channel } & \text{apex}_\text{req}_\text{new}_\text{partition}_\text{mode} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
& \times \text{OPERATING}\_\text{MODE}\_\text{TYPE} \\
\text{channel } & \text{apex}_\text{req}_\text{partition}_\text{status} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
\text{channel } & \text{apex}_\text{get}_\text{partition}_\text{status} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
& \times \text{PARTITION}\_\text{STATUS}\_\text{TYPE} \\
\text{channel } & \text{apex}_\text{req}_\text{system}_\text{time} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
\text{channel } & \text{apex}_\text{get}_\text{system}_\text{time} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
& \times \text{SYSTEM}\_\text{TIME}\_\text{TYPE} \\
\text{channel } & \text{apex}_\text{req}_\text{partition}_\text{mode} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
\text{channel } & \text{apex}_\text{get}_\text{partition}_\text{mode} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
& \times \text{OPERATING}\_\text{MODE}\_\text{TYPE} \\
\text{channel } & \text{apex}_\text{new}_\text{partition}_\text{mode}_\text{fail} : \text{PARTITION}\_\text{ID}\_\text{TYPE}
\end{align*}
\]

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

\[
\begin{align*}
\text{channel } & \text{apex}_\text{send}_\text{mpHM}_\text{table} : \text{iseq}_1 [ \text{A653}\_\text{MultiPartitionHMTable}\text{Type} ] \\
& \times (\text{iseq}_1 \text{A653}\_\text{ErrorIdentifier}\text{Type})
\end{align*}
\]

5. Channels for communication between Partitions and the APEX

\[
\begin{align*}
\text{channel } & \text{part}_\text{set}_\text{partition}_\text{mode} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
& \times \text{ARINC}\_\text{CONSTANTS} \\
\text{channel } & \text{part}_\text{req}_\text{partition}_\text{status} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
\text{channel } & \text{part}_\text{get}_\text{partition}_\text{status} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
& \times \text{PARTITION}\_\text{STATUS}\_\text{TYPE} \\
\text{channel } & \text{part}_\text{req}_\text{system}_\text{time} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
\text{channel } & \text{part}_\text{get}_\text{system}_\text{time} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
& \times \text{SYSTEM}\_\text{TIME}\_\text{TYPE} \\
\text{channel } & \text{part}_\text{req}_\text{sampling}_\text{port}_\text{id} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
& \times \text{SAMPLING}\_\text{PORT}\_\text{NAME}\_\text{TYPE} \\
\text{channel } & \text{part}_\text{get}_\text{sampling}_\text{port}_\text{id} : \text{PARTITION}\_\text{ID}\_\text{TYPE} \\
& \times \text{SAMPLING}\_\text{PORT}\_\text{ID}\_\text{TYPE} \\
\text{channel } & \text{partHM}_\text{restart}_\text{partition} : \text{PARTITION}\_\text{ID}\_\text{TYPE}
\end{align*}
\]
6. Channels for communication between each Partition and its health monitor

channel part_send_partHM_table : PARTITION_ID_TYPE
   × A653_PartitionHMTableType
   × (seq1 A653_ErrorIdentifierType)

7. Channels for communication between ARINC processes and its Partition

channel proc_req_system_time : PARTITION_ID_TYPE
   × PROCESS_ID_TYPE
channel proc_get_system_time : PARTITION_ID_TYPE
   × PROCESS_ID_TYPE
   × SYSTEM_TIME_TYPE
channel proc_set_partition_mode : PARTITION_ID_TYPE
   × PROCESS_ID_TYPE
   × ARINC_CONSTANTS
channel proc_req_sampling_port_id : PARTITION_ID_TYPE
   × PROCESS_ID_TYPE
   × SAMPLING_PORT_NAME_TYPE
channel proc_get_sampling_port_id : PARTITION_ID_TYPE
   × PROCESS_ID_TYPE
   × SAMPLING_PORT_ID_TYPE
channel proc_req_partition_status : PARTITION_ID_TYPE
   × PROCESS_ID_TYPE
channel proc_get_partition_status : PARTITION_ID_TYPE
   × PROCESS_ID_TYPE
   × PARTITION_STATUS_TYPE

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

channel os_send_modHM_table : iseq1 [A653_ModuleHMTableType]
   × seq1 A653_ErrorIdentifierType

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

channel updateClock : SYSTEM_TIME_TYPE
A.3 Partitions

A.3.1 Channels

\[
\text{channelset} \text{PartitionApex} == \{ \text{part}_\text{req}_\text{partition_status}, \text{part}_\text{get}_\text{partition_status}, \text{part}_\text{req}_\text{sampling_port_id}, \text{part}_\text{get}_\text{sampling_port_id}, \text{part}_\text{req}_\text{system_time}, \text{part}_\text{get}_\text{system_time}, \text{return}_\text{code}, \text{part}_\text{set}_\text{partition_mode}, \text{module}_\text{Init}, \text{module}_\text{End}, \text{module}_\text{EndInit}, \text{end}_\text{Partition}, \text{reinit}_\text{Partition} \}
\]

A.3.2 Partition process

\[
\text{process} \ A653\_\text{Partition} \triangleq \text{partitionId} : \text{PARTITION\_ID\_TYPE}; \\
\text{partHM} : \ A653\_\text{PartitionHMTableType}; \\
\text{sysError} : \text{seq} \ A653\_\text{ErrorIdentifierType} \bullet \text{begin}
\]

\text{GetTime} \triangleq \\
\text{proc}_\text{req}_\text{system_time}.\text{partitionId}?\text{processId} \rightarrow \\
\text{part}_\text{req}_\text{system_time}.\text{partitionId} \rightarrow \\
\text{part}_\text{get}_\text{system_time}.\text{partitionId}?t \rightarrow \\
\text{proc}_\text{get}_\text{system_time}.\text{partitionId}.\text{processId}!t \rightarrow \\
\text{return}_\text{code}.\text{partitionId}?\text{rc} \rightarrow \text{Skip}

\text{SetPartitionMode} \triangleq \\
\text{proc}_\text{set}_\text{partition_mode}.\text{partitionId}?\text{processId}?\text{OPERATING\_MODE} \rightarrow \\
\text{part}_\text{set}_\text{partition_mode}.\text{partitionId}!\text{OPERATING\_MODE} \rightarrow \\
\text{return}_\text{code}.\text{partitionId}?\text{rc} \rightarrow \text{Skip}

\text{GetSamplingPortId} \triangleq \\
\text{proc}_\text{req}_\text{sampling_port_id}.\text{partitionId}?\text{processId}?\text{SAMPLING\_PORT\_NAME} \rightarrow \\
\text{part}_\text{req}_\text{sampling_port_id}.\text{partitionId}!\text{SAMPLING\_PORT\_NAME} \rightarrow \\
\text{part}_\text{get}_\text{sampling_port_id}.\text{partitionId}?\text{spid} \rightarrow \\
\text{proc}_\text{get}_\text{sampling_port_id}.\text{partitionId}!\text{processId}!\text{spid} \rightarrow \\
\text{return}_\text{code}.\text{partitionId}?\text{rc} \rightarrow \text{Skip}
A.3. PARTITIONS

GetPartitionStatus ≜
proc_req_partition_status.partitionId?processId →
part_req_partition_status.partitionId →
part_get_partition_status.partitionId?st →
proc_get_partition_status.partitionId.processId!st →
return_code.partitionId?rc → Skip

ExecPartitionServices ≜
execPartition.partitionId →
\( \mu X \bullet \left( \begin{array}{c}
\text{SetPartitionMode} \\
\square \text{GetPartitionStatus} \\
\square \text{GetSamplingPortId} \\
\square \text{GetTime} \\
\triangle (\text{interruptPartition.partitionId} \rightarrow \text{Skip})
\end{array} \right) ; X \)

ExecPartition ≜ initPartition.partitionId →
part_send_partHM_table.partitionId!pHM!sysError →
moduleEndInit →
\( \left( \mu X \bullet \text{ExecPartitionServices} ; X \right) \)
\( \triangle \left( \text{endPartition.partitionId} \rightarrow \text{Skip} \right) \)
\( \square \text{reinitPartition.partitionId} \rightarrow \text{ReinitPartition} \)

ReinitPartition ≜ partHM_restart_partition.partitionId →
\( \left( \mu X \bullet \text{ExecPartitionServices} ; X \right) \)
\( \triangle \left( \text{endPartition.partitionId} \rightarrow \text{Skip} \right) \)
\( \square \text{reinitPartition.partitionId} \rightarrow \text{ReinitPartition} \)

\( \bullet \text{moduleInit} \rightarrow (\text{ExecPartition} \triangle \text{moduleEnd} \rightarrow \text{Skip}) \)

end

A.3.3 The Partition Level Health Monitor

process PartitionHM ≜ partitionId : PARTITION_ID_TYPE \bullet \text{begin}


\[
\text{state } \text{PartitionHMSt} = \{\text{pHM : A653\_PartitionHMTableType; sysError : seq1 A653\_ErrorIdentifierType}\}
\]

\[
\bullet \left( \text{part\_send\_partHM\_table.partitionId?phm?se \rightarrow} \right)
\]

\[
\text{end}
\]

\[
\text{channelset } \text{PartitionHMChannels} = \{ \text{part\_send\_partHM\_table, moduleEnd, endPartition} \}
\]

### A.3.4 Partitions Layer Model

\[
\text{process } \text{Partitions\_Layer} \equiv \text{partitionIds : P PARTITION\_ID\_TYPE; partHM : seq1 A653\_PartitionHMTableType; sysError : seq1 A653\_ErrorIdentifierType}
\]

\[
\bullet \left( \| \text{pid : partitionIds} \| \{ \text{moduleInit, moduleEnd, moduleEndInit} \} \right)
\]

\[
\bullet \left( \text{A653\_Partition(pid, partHM(pid), sysError)} \right)
\]

\[
\left\{ \text{PartitionHMChannels} \| \text{PartitionHM(pid)} \right\}
\]

### A.4 The APEX

\[
\text{channelset } \text{ApexOs} = \{ \text{moduleInit, moduleEnd, moduleEndInit, endPartition, reinitPartition, return\_code, apex\_get\_sampling\_port\_id, apex\_req\_new\_partition\_mode, apex\_req\_partition\_status, apex\_get\_partition\_status, apex\_req\_system\_time, apex\_get\_system\_time, apex\_req\_partition\_mode, apex\_get\_partition\_mode, apex\_req\_sampling\_port\_id, apex\_new\_partition\_mode\_fail} \}
\]

### A.4.1 APEX Process

\[
\text{process } \text{APEX} \equiv \text{sysError : seq1 A653\_ErrorIdentifierType; mpHM : seq1 A653\_MultiPartitionHMTableType} \bullet \text{begin}
\]
\[ GET\_TIME \triangleq part\_req\_system\_time?pid\rightarrow \]
\[ apex\_req\_system\_time.pid\rightarrow \]
\[ apex\_get\_system\_time.pid?t\rightarrow \]
\[ part\_get\_system\_time.pid!t\rightarrow \]
\[ return\_code.pid!NO\_ERROR \rightarrow \text{Skip} \]

\[ GET\_PARTITION\_STATUS \triangleq \]
\[ part\_req\_partition\_status?pid\rightarrow \]
\[ apex\_req\_partition\_status.pid\rightarrow \]
\[ apex\_get\_partition\_status.pid?cp\rightarrow \]
\[ part\_get\_partition\_status.pid!cp \rightarrow \text{Skip} \]

\[ SET\_PARTITION\_MODE \triangleq \]
\[ part\_set\_partition\_mode?pid\_nopm\rightarrow \]
\[ apex\_req\_partition\_mode.pid\rightarrow \]
\[ apex\_get\_partition\_mode.pid?opm\rightarrow \]
\[ \begin{cases} 
\text{if } (nopm \notin \text{OPERATING\_MODE\_TYPE}) \rightarrow \\
\quad \text{return\_code.pid!INVALID\_PARAM \rightarrow Skip} \\
\quad (opm = \text{NORMAL} \land nopm = \text{NORMAL}) \rightarrow \\
\quad \quad \text{apex\_new\_partition\_mode\_fail.pid} \rightarrow \\
\quad \quad \text{return\_code.pid!NO\_ACTION \rightarrow Skip} \\
\quad (opm = \text{COLD\_START} \land nopm = \text{WARM\_START}) \rightarrow \\
\quad \quad \text{apex\_new\_partition\_mode\_fail.pid} \rightarrow \\
\quad \quad \text{return\_code.pid!INVALID\_MODE \rightarrow Skip} \\
\quad (nopm = \text{IDLE} \\
\quad \lor \ nopm \in \text{OPERATING\_INIT\_MODE\_TYPE}) \rightarrow \\
\quad \quad \text{apex\_req\_new\_partition\_mode.pid!nopm} \rightarrow \\
\quad \quad \text{return\_code.pid!NO\_ERROR \rightarrow Skip} \\
\quad (opm \neq \text{NORMAL} \land nopm = \text{NORMAL}) \rightarrow \\
\quad \quad \text{apex\_req\_new\_partition\_mode.pid!nopm} \rightarrow \\
\quad \quad \text{return\_code.pid!NO\_ERROR \rightarrow Skip} \\
\end{cases} \]
GET_SAMPLING_PORT_ID ≜
part_req_sampling_port_id?pid?spn→
apex_req_sampling_port_id.pid!spn→
apex_get_sampling_port_id.pid?spid→
   if (spid = -1)→
      part_get_sampling_port_id.pid!spid→
      return_code.pid!INVALID_CONFIG → Skip
   [] (spid ≠ -1)→
      part_get_sampling_port_id.pid!spid→
      return_code.pid!NO_ERROR → Skip
fi

ExecApexServices ≜ execPartition?pid→
   \( \mu X \bullet \) \( \begin{align*}
   \Box \text{GET\_TIME} \\
   \Box \text{SET\_PARTITION\_MODE} \\
   \Box \text{GET\_PARTITION\_STATUS} \\
   \Box \text{GET\_SAMPLING\_PORT\_ID} \\
   \text{interruptPartition.pid} \rightarrow \text{Skip} \\
   \Box \text{endPartition.pid} \rightarrow \text{Skip}
   \end{align*} \) ; X

ExecApex ≜ apex_send_mpHM_table!mpHM!sysError→
moduleEndInit → (\( \mu X \bullet \text{ExecApexServices} \); X)

• moduleInit → (ExecApex ⋁ moduleEnd → Skip)

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

process MultiPartitionHM ≜ begin

state MultiPartitionHMS t ==
   [mpHM : seq_1 A653_MultiPartitionHTableType;
    sysError : seq_1 A653_ErrorIdentifierType]
• apex_send_mpHM_table?mp?se→
   mpHM, sysError := mp, se
   △moduleEnd → Skip

end
A.5. THE OPERATING SYSTEM

channelset MultiPartitionHMChannels == ⦃ apex_send_mpHM_table, moduleEnd ⦄

A.4.3 APEX Layer Model

\[
\text{process } APEX\_Layer \triangleq mpHM : \text{seq}_1 \text{A653\_MultiPartitionHMTableType}; \\
\text{sysError} : \text{seq}_1 \text{A653\_ErrorIdentifierType} \\
\quad \cdot \left( \text{APEX}(\text{sysError}, mpHM) \right) \\
\quad \left[ \text{MultiPartitionHMChannels} \right] \text{MultiPartitionHM}
\]

A.5 The Operating System

A.5.1 The Operating System Circus process

\[
\text{process } OperatingSystem \triangleq \text{module} : \text{Module} \bullet \text{begin} \\
\begin{array}{|l|}
\hline
\text{partsions\_variables} : \text{iseq}[\text{PartitionVariables}] \\
\text{major\_time\_frame} : \text{SYSTEM\_TIME\_TYPE} \\
\text{system\_time} : \text{SYSTEM\_TIME\_TYPE} \\
\text{current\_partition} : \text{A653\_PartitionTimeWindowType} \\
\text{current\_partition\_id} : \text{PARTITION\_ID\_TYPE} \\
\text{sampling\_ports} : (\text{SAMPLING\_PORT\_ID\_TYPE} \rightarrow \text{SAMPLING\_PORT\_NAME\_TYPE}) \\
\text{queuing\_ports} : (\text{QUEUING\_PORT\_ID\_TYPE} \rightarrow \text{QUEUING\_PORT\_NAME\_TYPE}) \\
\hline
\end{array}
\]

\[
\# \text{partsions\_variables} = \# \text{module\_Partitions}
\]

state OSSt

\[
\text{InitPartition} \triangleq \left\| \left\| \text{pid} : (1 \ldots (\# \text{module\_Partitions})) \right\| \text{initPartition.pid} \rightarrow \text{Skip} \right\|
\]
**MajorTimeFrame**

\[ \Delta \text{OSS}t \]

\[ \text{getMajorTimeFrame} : \text{PartitionsType} \]

\[ \forall x : \text{ran module.Partitions} \]
\[ \bullet \# (\text{module.Partitions} \triangleright \{x\}) = \# (\text{getMajorTimeFrame} \triangleright \{x\}) \]

\[ \forall p1, p2 : \text{ran getMajorTimeFrame} \]
\[ \bullet p1.\text{PartitionPeriodicity.Period} > p2.\text{PartitionPeriodicity.Period} \]

\[ \text{major_time_frame}' = (\text{getMajorTimeFrame}(1)).\text{PartitionPeriodicity.Period} \]

\[ \theta(\text{OSS}t \setminus (\text{major_time_frame}'))' = \theta(\text{OSS}t \setminus (\text{major_time_frame})) \]

**NextPartition**

\[ \Delta \text{OSS}t \]

\[ \text{current_partition}' = \text{head} (\text{module.Schedules} \mid \]
\[ \{ p : \text{A653_PartitionTimeWindowType} \mid \]
\[ (\text{system_time mod major_time_frame}) \leq p.\text{Offset}) \}
\[ (\text{module.Partitions(current_partition_id')}).\text{PartitionDefinition.Name} = \]
\[ (\text{current_partition}').\text{PartitionNameRef} \]

\[ \theta(\text{OSS}t \setminus (\text{current_partition_id}, \text{current_partition}'))' = \]

\[ \theta(\text{OSS}t \setminus (\text{current_partition_id}, \text{current_partition})) \]

\[ \text{UpdateSystemTime} \triangleq \text{updateClock}?x \rightarrow \text{system_time} := x \]

\[ \text{OSGetTime} \triangleq \text{apex_req_system_time.current_partition_id} \rightarrow \text{UpdateSystemTime}; \]
\[ \text{apex_get_system_time.current_partition_id!system_time} \rightarrow \text{Skip} \]
\text{PartitionStatus} = \exists \text{OSSt}
\begin{align*}
\mathbf{p}! : \text{PARTITION STATUS TYPE} \\
(p!).\text{Identifier} = \\
\quad (\text{module.Partitions(current_partition_id)}).\text{PartitionDefinition.Identifier} \\
(p!).\text{Period} = \\
\quad (\text{module.Partitions(current_partition_id)}).\text{PartitionPeriodicity.Period} \\
(p!).\text{Duration} = \\
\quad (\text{module.Partitions(current_partition_id)}).\text{PartitionPeriodicity.Duration} \\
(p!).\text{LOCK LEVEL} = \\
\quad (\text{partitions.variables(current_partition_id)}).\text{LOCK LEVEL} \\
(p!).\text{OPERATING MODE} = \\
\quad (\text{partitions.variables(current_partition_id)}).\text{OPERATING MODE} \\
(p!).\text{START CONDITION} = \\
\quad (\text{partitions.variables(current_partition_id)}).\text{START CONDITION}
\end{align*}

\text{OSGetPartitionStatus} \triangleq \var p : \text{PARTITION STATUS TYPE} \bullet \\
\text{apex.req.partition_status.current_partition_id} \rightarrow (\text{PartitionStatus}); \\
\text{apex.get.partition_status.current_partition_id!p} \rightarrow \text{Skip}

\text{SetPMode} \\
\triangleq \var \text{OSSt} \\
\text{nopm}! : \text{OPERATING MODE TYPE} \\
\text{partitions.variables}' = \\
\quad (\text{let} \ o = \langle \text{OPERATING MODE} = \text{nopm}!, \ \\
\text{START CONDITION} = \\
\quad (\text{partitions.variables(current_partition_id)}).\text{START CONDITION}, \ \\
\text{LOCK LEVEL} = \\
\quad (\text{partitions.variables(current_partition_id)}).\text{LOCK LEVEL}, \\
\quad \bullet \text{partitions.variables} \oplus \{\text{current_partition_id} \mapsto o\}) \ \\
\theta(\text{OSSt} \setminus \{\text{current_partition_id, current_partition, partitions.variables}\})' = \\
\quad \theta(\text{OSSt} \setminus \{\text{current_partition_id, current_partition, partitions.variables}\})$
\[\text{OSSetPartitionMode} \triangleq \text{var } \text{opm} : \text{OPERATING\_MODE\_TYPE} \bullet\]

\[
\begin{align*}
\text{apex\_req\_partition\_mode}\cdot\text{current\_partition\_id} & \rightarrow \\
\text{apex\_get\_partition\_mode}\cdot\text{current\_partition\_id}!((\text{PartitionsVariables}\cdot\text{current\_partition\_id}), \text{OPERATING\_MODE}) & \rightarrow \\
\left\{
\begin{array}{l}
\text{if} (\text{nopm} = \text{IDLE}) \rightarrow \\
\quad \text{endPartition}\cdot\text{current\_partition\_id} \rightarrow (\text{SetPMode}) \\
\quad \text{if} (\text{nopm} \in \text{OPERATING\_INIT\_MODE\_TYPE}) \rightarrow \\
\quad \quad \text{reinitPartition}\cdot\text{current\_partition\_id} \rightarrow (\text{SetPMode}) \\
\quad \text{fi} \\
\end{array}
\right.
\end{align*}
\]

\[\text{OGetSamplingPortId} \triangleq \]

\[
\begin{align*}
\text{apex\_req\_sampling\_port\_id}\cdot\text{current\_partition\_id}\cdot\text{spn} & \rightarrow \\
\left\{
\begin{array}{l}
\text{if} (\text{spn} \in \text{ran sampling\_ports}) \rightarrow \\
\quad \text{apex\_get\_sampling\_port\_id}\cdot\text{current\_partition\_id}!(\mu \\
\quad \quad \text{x} : (\text{dom(sampling\_ports} \triangleright \{\text{spn}\})) \bullet \text{x}) \rightarrow \\
\quad \quad \text{return\_code}\cdot\text{current\_partition\_id}!\text{NO\_ERROR} \rightarrow \text{Skip} \\
\quad \text{if} (\text{spn} \notin \text{ran sampling\_ports}) \rightarrow \\
\quad \quad \text{apex\_get\_sampling\_port\_id}\cdot\text{current\_partition\_id}!(\text{-1}) \rightarrow \\
\quad \quad \text{return\_code}\cdot\text{current\_partition\_id}!\text{INVALID\_CONFIG} \rightarrow \text{Skip} \\
\end{array}
\right.
\end{align*}
\]

\[\text{ExecOSServices} \triangleq \]

\[
\begin{align*}
\text{execPartition}\cdot\text{current\_partition\_id} & \rightarrow \\
\left(\mu X \bullet \\
\begin{array}{c}
\text{OGetTime} \\
\text{OGetPartitionStatus} \\
\text{OSetPartitionMode} \\
\text{OGetSamplingPortId}
\end{array} ; X \right)
\end{align*}
\]
A.5. THE OPERATING SYSTEM

\[ \text{ExecOS} \triangleq \text{var getMajorTimeFrame : PartitionsType} \bullet (\text{MajorTimeFrame}); \]
\[
\left( \mu X \bullet \text{UpdateSystemTime} ; (\text{NextPartition}) ; \right)
\]
\[
\left( \begin{array}{l}
\text{if} \left( \text{current\_partition\_Offset} > \left( \text{system\_time mod major\_time\_frame} \right) \right) \rightarrow \\
\quad \text{wait} \left( \text{current\_partition\_Offset} - \left( \text{system\_time mod major\_time\_frame} \right) \right); X \\
\quad \text{if} \left( \text{system\_time mod major\_time\_frame} \right) \rightarrow \\
\quad \quad \text{ExecOSServices} \triangleright \\
\quad \quad \text{interruptPartition.current\_partition\_id}; X
\end{array} \right)
\]

\* \text{moduleInit} \rightarrow \\
\quad \text{os\_send\_modHM\_table!}(\text{module\_HealthMonitoring\_ModuleHM})!(\text{module\_HealthMonitoring\_SystemErrors}) \rightarrow (\text{ExecOS} \\
\quad \triangle \text{moduleEnd} \rightarrow \text{Skip})

end

A.5.2 The Timer

process \text{Timer} \triangleq \begin{array}{l}
\text{state} \: \text{TimerSt} \: = \: [\text{clock} : \text{SYSTEM\_TIME\_TYPE}] \\
\text{Counter} \triangleq \left( \mu X \bullet \left( \begin{array}{l}
(\text{wait}(1) ; \: \text{clock} \: := \: \text{clock} + 1) \\
\: \| \left( \mu Y \bullet \text{updateClock!clock} \rightarrow Y \right) \: \triangleright \: \text{Skip}
\end{array} \right) ; X \right)
\end{array}

\* \text{moduleInit} \rightarrow (\text{Counter} \: \triangle \: \text{moduleEnd} \rightarrow \text{Skip})

end

A.5.3 The Operating System Health Monitor (Module)

process \text{ModuleHM} \triangleq \begin{array}{l}
\end{array}
\textbf{state} \( \text{ModuleHMSt} \equiv [\text{modHM} : \text{seq}_1 \text{A653\_ModuleHMTableType}; \)
\[
\begin{align*}
\text{sysError} & : \text{seq}_1 \text{A653\_ErrorIdentifierType}; \\
\bullet (\text{os\_send\_modHM\_table?mp?se} \rightarrow \text{modHM}, \text{sysError} := \text{mp}, \text{se}) \\
\end{align*}
\]

\( \triangle \text{moduleEnd} \rightarrow \text{Skip} \)

\textbf{end}

\textbf{channelset} \( \text{ModuleHMChannels} \equiv \{ \text{os\_send\_modHM\_table}, \text{moduleEnd} \} \)

A.5.4 The Operating System Layer Model

\textbf{process} \( \text{OS\_Layer} \equiv \text{module} : \text{Module} \)
\[
\begin{align*}
( & \text{OperatingSystem}(\text{module}) \) \\
\text{Timer} & \left( \text{ModuleHMChannels} \parallel \text{ModuleHM} \right) \\
\end{align*}
\]

\textbf{process} \( \text{IMA\_Module} \equiv \text{module} : \text{Module} \)
\[
\begin{align*}
( & \text{OS\_Layer}(\text{module}) \) \\
\text{ApexOs} & \left( \text{module.\text{HealthMonitoring}.MultiPartitionHM}, \text{module.\text{HealthMonitoring}.SystemErrors} \right) \\
\text{APEX\_Layer} & \left( \text{module.\text{HealthMonitoring}.PartitionHM}, \text{module.\text{HealthMonitoring}.SystemErrors} \right) \\
\text{PartitionApex} & \left( \text{1..\#module.\text{Partitions}}, \text{module.\text{HealthMonitoring}.PartitionHM}, \text{module.\text{HealthMonitoring}.SystemErrors} \right) \\
\text{Partitions\_Layer} & \left( \text{1..\#module.\text{Partitions}}, \text{module.\text{HealthMonitoring}.PartitionHM}, \text{module.\text{HealthMonitoring}.SystemErrors} \right) \\
\end{align*}
\]
Bibliography


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


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


