Universidade do Minho
Departamento de Informática

Ricardo Jorge Cantador Romano

Formal approaches to critical systems development: a case study using SPARK

Mestrado em Engenharia Informática

Trabalho efetuado sob a orientação do
Professor Doutor Manuel Alcino Cunha

Outubro de 2011
Acknowledgements

First of all, I would like to extend my sincere appreciation to Prof. Alcino Cunha for having made himself available to supervise this dissertation. His commitment, willingness and surgical comments were of most value and essential for the completion of this work.

Part of this work was elaborated at Critical Software company, and hence I’m grateful for the excellent human and logistic conditions provided. I’d like to thank all the people the good atmosphere and the permanent contagious good mood, it was indeed extremely rewarding working in such a friendly environment. In particular, I want to express my gratitude to my local supervisor and friend, José Faria, for always being available to give me a hand; also, to André and Pedro - the FM crew - who were authentic companions of work and leisure, and to Pedrosa who devoted much of his time to many lively and fruitful discussions.

To all the inspiring lecturers I had throughout my academic walk, in particular to Abel Gomes, António Sousa, António Nestor, Gâel Dias, Hugo Proença, José Creissac, José Nuno Oliveira, Luís Alexandre, Luís Barbosa, Paulo Moura, Sara Madeira. A special reference must go to Simão Melo de Sousa for both introducing to me the formal “stuff” and his friendship along the past few years.

To my friends, Diana, Eduardo, Flávio, and Marco, I know I don’t see you as often as I’d like to, but I’m pretty much sure you’re always out there helping me out to keep myself sane.

To my beloved family, my parents, João and Esperança, my aunts, Cilinha and Helena, and my grandfather “Amendoim” for all their support, confidence deposited on me, and for so strongly encouraging me.“Muito obrigado, devo-vos tudo!”

To Sara, my half-orange, for all her love and total dedication, and to my adoptive family, Maria Manuel, dona Mimi, Pim Preta, and Zé Paulo for putting up with me so happily in recent times. A special word to sr. Zé for his friendship and wise advices.

During the elaboration of this work many other people have supported me either directly or indirectly. Unfortunately I’m not able to mention everyone in here, but still I’m also grateful to all of them.
Abstract

Formal approaches to critical systems development: a case study using SPARK

Formal methods comprise a wide set of languages, technologies and tools based on mathematics (logic, set theory) for specification, development and validation of software systems. In some domains, its use is mandatory for certification of critical software components requiring high assurance levels [1][2].

In this context, the aim of this work is to evaluate, in practice and using SPARK [3], the usage of formal methods, namely the "Correctness by Construction" paradigm [4], in the development of critical systems. SPARK is a subset of Ada language that uses annotations (contracts), in the form of Ada comments, which describe the desired behavior of the component.

Our case study is a microkernel of a real-time operating system based on MILS (Multiple Independent Levels of Security/Safety) architecture [5]. It was formally developed in an attempt to cover the security requirements imposed by the highest levels of certification.
Resumo

Os métodos formais agregam todo um conjunto de linguagens, tecnologias e ferramentas baseadas em matemática (lógica, teoria de conjuntos) para a especificação, desenvolvimento e validação de sistemas de software. A sua utilização é, em certos domínios, inclusivamente obrigatória para a certificação de componentes de software crítico segundo os níveis mais elevados de segurança [1, 2].

Neste contexto, pretende-se com este trabalho avaliar em termos práticos e com o uso do SPARK [3], a utilização dos métodos formais, nomeadamente o paradigma "Correctness by Construction" [4], no desenvolvimento de sistemas críticos. A linguagem SPARK consiste num subconjunto da linguagem Ada, que utiliza anotações (contractos), sob a forma de comentários em Ada, que descrevem o comportamento desejado do componente.

O caso de estudo consiste num microkernel de um sistema operativo de tempo real baseado na arquitectura MILS (Multiple Independent Levels of Security/Safety) [5]. A sua modelação procurou cobrir os requisitos de segurança impostos pelos mais elevados níveis de certificação.
Contents

Abstract v

Resumo vii

List of Figures xii

List of Tables xiii

Code Listings xv

1 Introduction 1

1.1 Critical systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Operating system . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.3 Our aim and objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.4 Contribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.5 Dissertation outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2 High assurance software development with SPARK 5

2.1 Formal methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.1.1 Classical . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.1.2 Lightweight . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.1.3 Directed to the code . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.2 SPARK overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.3 Development methodology . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.3.1 Correctness by construction . . . . . . . . . . . . . . . . . . . . . . . 20
2.3.1.1 Formal specification . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.3.1.2 INFORMED . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.3.2 Tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
2.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

3 Case study: a secure partitioning microkernel 29

3.1 Microkernel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
## Contents

3.2 Separation kernel .................................................. 32
   3.2.1 Spatial partitioning ........................................... 33
   3.2.2 Temporal partitioning ......................................... 34
3.3 Security partitioning and MILS ................................... 34
3.4 Summary ............................................................ 39

4 Formal approaches to kernel development ....................... 40
   4.1 Traditional kernels ............................................... 41
   4.2 Separation kernels ............................................... 49
   4.3 Tokeneer project .................................................. 52
   4.4 Summary ........................................................... 53

5 Implementation ....................................................... 54
   5.1 Formal specification/design ..................................... 55
   5.2 INFORMED in practice ............................................ 58
      5.2.1 Identification of the system boundary, inputs and outputs ... 58
      5.2.2 Identification of the SPARK boundary ....................... 59
      5.2.3 Identification and localization of system states .......... 59
   5.3 Handling initialization of state ................................... 61
   5.4 Handling secondary requirements ................................ 61
   5.5 Implementing the internal behavior of components ............ 62
   5.6 Other issues ....................................................... 73

6 Conclusions .......................................................... 78
   6.1 Main aim and objectives .......................................... 78
   6.2 Contributions ..................................................... 81
   6.3 Future work ....................................................... 82

Bibliography ............................................................ 83

A Requirements ........................................................ 97
   A.1 Configuration management system requirements ................ 98
   A.2 System error and faults requirements .......................... 100
   A.3 System table requirements ...................................... 101
   A.4 Partition requirements .......................................... 102
   A.5 Process requirements ............................................ 105
   A.6 Partition information flow requirements ........................ 107

B Z models ............................................................. 109
   B.1 Types ............................................................. 109
      B.1.1 Memory ....................................................... 109
      B.1.2 Kernel ....................................................... 110
List of Figures

2.1 Ada and SPARK .................................................. 14
2.2 INFORMED notation ............................................. 24
2.3 Identification of the system boundary .......................... 25
2.4 Identification of the SPARK boundary ......................... 25

3.1 Monolithic kernel and microkernel ............................. 31
3.2 Two level scheduler ............................................... 35
3.3 Multiple Independent Levels of Safety/Security (MILS) architecture .................................................. 36
3.4 Inter-partition communication .................................. 37
3.5 Policies of communication ...................................... 38

5.1 Implementation steps ............................................ 55
5.2 SPARK system boundary ........................................ 59
5.3 Localization of state ............................................. 61
5.4 A possible state of communication memory .................... 75
5.5 POGS summary .................................................. 77
# List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Time, cost and quality with use of Formal Methods</td>
<td>6</td>
</tr>
<tr>
<td>4.1</td>
<td>Traditional kernels verification</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>System boundary, inputs and outputs</td>
<td>58</td>
</tr>
<tr>
<td>5.2</td>
<td>State identification</td>
<td>60</td>
</tr>
<tr>
<td>5.3</td>
<td>PartitionTypes package</td>
<td>63</td>
</tr>
<tr>
<td>5.4</td>
<td>PRT package</td>
<td>64</td>
</tr>
<tr>
<td>5.5</td>
<td>TablesTypes package</td>
<td>66</td>
</tr>
<tr>
<td>5.6</td>
<td>ErrorTypes package</td>
<td>68</td>
</tr>
<tr>
<td>5.7</td>
<td>SEF package</td>
<td>69</td>
</tr>
<tr>
<td>5.8</td>
<td>Hardware package</td>
<td>70</td>
</tr>
<tr>
<td>5.9</td>
<td>CMS package</td>
<td>71</td>
</tr>
<tr>
<td>5.10</td>
<td>SYT package</td>
<td>72</td>
</tr>
<tr>
<td>5.11</td>
<td>Configuration file format</td>
<td>74</td>
</tr>
<tr>
<td>A.1</td>
<td>Requirement format</td>
<td>98</td>
</tr>
<tr>
<td>A.2</td>
<td>CMS requirement # 01</td>
<td>98</td>
</tr>
<tr>
<td>A.3</td>
<td>CMS requirement # 02</td>
<td>99</td>
</tr>
<tr>
<td>A.4</td>
<td>CMS requirement # 03</td>
<td>99</td>
</tr>
<tr>
<td>A.5</td>
<td>CMS requirement # 04</td>
<td>99</td>
</tr>
<tr>
<td>A.6</td>
<td>CMS requirement # 05</td>
<td>100</td>
</tr>
<tr>
<td>A.7</td>
<td>SEF requirement # 01</td>
<td>100</td>
</tr>
<tr>
<td>A.8</td>
<td>SEF requirement # 02</td>
<td>100</td>
</tr>
<tr>
<td>A.9</td>
<td>SYT requirement # 01</td>
<td>101</td>
</tr>
<tr>
<td>A.10</td>
<td>SYT requirement # 02</td>
<td>101</td>
</tr>
<tr>
<td>A.11</td>
<td>SYT requirement # 03</td>
<td>101</td>
</tr>
<tr>
<td>A.12</td>
<td>PRT requirement # 01</td>
<td>102</td>
</tr>
<tr>
<td>A.13</td>
<td>PRT requirement # 02</td>
<td>102</td>
</tr>
<tr>
<td>A.14</td>
<td>PRT requirement # 03</td>
<td>102</td>
</tr>
<tr>
<td>A.15</td>
<td>PRT requirement # 04</td>
<td>103</td>
</tr>
<tr>
<td>A.16</td>
<td>PRT requirement # 05</td>
<td>103</td>
</tr>
<tr>
<td>A.17</td>
<td>PRT requirement # 06</td>
<td>103</td>
</tr>
<tr>
<td>A.18</td>
<td>PRT requirement # 07</td>
<td>104</td>
</tr>
<tr>
<td>Requirement #</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>A.19 PRT</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>A.20 PRT</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>A.21 PRT</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>A.22 PRT</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>A.23 PRC</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>A.24 PRC</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>A.25 PRC</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>A.26 PRC</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>A.27 PRC</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>A.28 PIFP</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>A.29 PIFP</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>A.30 PIFP</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>A.31 PIFP</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>A.32 PIFP</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>A.33 PIFP</td>
<td>108</td>
<td></td>
</tr>
</tbody>
</table>
## Code Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Stack specification package</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>Stack body package</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Push and Pop procedures with proof annotations</td>
<td>18</td>
</tr>
<tr>
<td>5.1</td>
<td>PRT specification package (GetID, GetDuration and GetPartitionMode functions)</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>PRT specification package (SetPartitionMode procedure)</td>
<td>65</td>
</tr>
<tr>
<td>5.3</td>
<td>PRT specification package (Init procedure)</td>
<td>65</td>
</tr>
<tr>
<td>5.4</td>
<td>PRT specification package (ClearCommunicationList procedure)</td>
<td>66</td>
</tr>
<tr>
<td>5.5</td>
<td>TablesTypes specification package</td>
<td>67</td>
</tr>
<tr>
<td>5.6</td>
<td>SEF specification package</td>
<td>70</td>
</tr>
<tr>
<td>5.7</td>
<td>Hardware specification package (CanAllocate function and AllocBlock, Init procedures)</td>
<td>71</td>
</tr>
<tr>
<td>C.1</td>
<td>DefaultValues package</td>
<td>131</td>
</tr>
<tr>
<td>C.2</td>
<td>PartitionTypes package</td>
<td>131</td>
</tr>
<tr>
<td>C.3</td>
<td>PRT specification package</td>
<td>133</td>
</tr>
<tr>
<td>C.4</td>
<td>PRT body package</td>
<td>134</td>
</tr>
<tr>
<td>C.5</td>
<td>TablesTypes package</td>
<td>135</td>
</tr>
<tr>
<td>C.6</td>
<td>HardwareTypes package</td>
<td>136</td>
</tr>
<tr>
<td>C.7</td>
<td>ErrorTypes package</td>
<td>137</td>
</tr>
<tr>
<td>C.8</td>
<td>SEF specification package</td>
<td>137</td>
</tr>
<tr>
<td>C.9</td>
<td>SEF body package</td>
<td>137</td>
</tr>
<tr>
<td>C.10</td>
<td>Hardware specification package</td>
<td>138</td>
</tr>
<tr>
<td>C.11</td>
<td>Hardware body package</td>
<td>138</td>
</tr>
<tr>
<td>C.12</td>
<td>CMS specification package</td>
<td>140</td>
</tr>
<tr>
<td>C.13</td>
<td>CMS body package</td>
<td>142</td>
</tr>
<tr>
<td>C.14</td>
<td>SYT specification package</td>
<td>143</td>
</tr>
<tr>
<td>C.15</td>
<td>SYT body package</td>
<td>146</td>
</tr>
<tr>
<td>C.16</td>
<td>ConfigValues specification package</td>
<td>156</td>
</tr>
<tr>
<td>C.17</td>
<td>ConfigValues body package</td>
<td>158</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In our modern society, software is everywhere. Examples of systems controlled by software can be found in sectors like aerospace, railway, banking, medical, energy, defense, among others. Many of these systems are critical systems whose failures threaten human lives. So, the correction of this software is a demand. Any error on safety critical systems can have catastrophic consequences, can originate loss of life or damage to the environment. On security systems an error may be equally devastator, as in the case of loss of national security or commercial reputation.

1.1 Critical systems

Safety critical and secure systems must be designed with great care, and the correctness of the software is highly important to guarantee their integrity. This kind of systems have a trusted set that consists in the hardware and software required to ensure the system security policy. These components, and the software is no exception, have some properties that are demanded for the behavior of the system to be considered safe and secure. In software, where the scope of this work is, these properties can be expressed as predicates that must be satisfied and maintained while the system works even with the inputs/outputs interferences.

For example, an aircraft cabin pressure control is of vital importance to the passengers and crew. This system ensures that the air pressure is maintained within predefined limits, taking a special care in the rapid changes of pressure to ensure occupant comfort.
It also protects the airplane itself against problems that might be caused by high difference between internal and external pressure. System sensors read the pressure value and pass those data on to the controllers to verify the need to increase or decrease the pressure inside the plane. This management should be done within a set up time interval to maintain the safety of occupants and the airplane itself. An aircraft cabin pressure control helps to keep the plane in a safe state. Above 3000 meters a pressurization system failure requires an emergency descent to 3000 meters for the availability of oxygen to its occupants and hence for the masks to come out.

In this example, safety is defined with respect to occupants security and good condition of airplane. The safety predicate for the software, besides of course that the system must run always without errors, is the time that elapses between the inputs from the sensors to the outputs of the actuators of the system that controls the air pressure. For safety purposes, this predicate for the pressurization control software needs to increase or decrease the pressure (to achieve safety values) inside of the aircraft at a range of initial time set up beforehand. As mentioned above, a failure of the aircraft cabin pressure control to be detected above 3000 meters requires a descent of the aircraft as well as the triggering device that releases automatically oxygen masks for passengers. To control all these devices and different associate systems we need an operating system (OS). The OS allows all components to run and perform its tasks besides allowing communication between them.

1.2 Operating system

Clearly the OS is a key component of the whole system, whose correctness and reliability depends on the OS. So, the OS is part of the trusted set of the system. Yet these systems with a security policy do not support the common OS (monolithic kernel), because they can not deal with correctness, reliability, and security. The microkernels adapt better to this type of systems due to the principles of least privilege and minimality used in the architecture. However, in order to make a reliable microkernel, a correct design is necessary and also a correct implementation of it. Those steps can be achieved with the use of formal methods in order to be able to verify their accuracy. This is in fact the main subject of this MSc work.
Chapter 1. Introduction

1.3 Our aim and objectives

The main aim of this dissertation is to study the usage of SPARK [3] language and its associate methodology in systems development. Following a Correctness by Construction (CbyC)-based methodology, it aims to confirm what was demonstrated by other authors (including [6, 7]) concerning the possibility of developing systems able to achieve high levels of certification. The CbyC methodology was used by Altran Praxis and was more widely disseminated in the Tokeneer project [8]. This project was a collaboration of Praxis and NSA (National Security Agency) to demonstrate that it is possible to develop systems up to the level of rigor required by the highest standards of the Common Criteria. In Chapter [4], we introduce the Tokeneer project in more detail.

Therefore, ultimately we sought to apply the Tokeneer project’s outcome in a different context, which constituted our case study. A kernel is surely a key central part of a system and therefore it was chosen to apply the methodology above. More specifically, a secure partitioning microkernel - a kernel with partitioning and security properties required for critical systems - was considered. Starting from a simplified but yet representative set of functional requirements for this type of system, we planed to develop a secure partitioning microkernel-based simulator using the CbyC development method.

In order to accomplish this, we identified the following steps to be performed:

- To investigate the system requirements;
- To use the Z notation to create a high level specification;
- To construct a design of the system with INFORMED process;
- To implement the system in SPARK;
- To verify the system using the SPARK Examiner toolset.

1.4 Contribution

We strongly believe that the work presented in this dissertation is a valid contribution in order to obtain a software capable of achieving the highest levels of certification, which is required when security systems are concerned. The subject under study is also of great relevance since it is a microkernel-based approach, which is clearly the central defining feature of an OS.
1.5 Dissertation outline

This dissertation is organized in six chapters. Chapter 1 introduces the main concepts that are subject of study in this work. Chapter 2 gives a brief overview about software engineering with formal methods and presents the SPARK language and the methodology followed in the work. Chapter 3 presents the work that is to be developed, giving an overview of what is a secure partitioning microkernel and the proposed solution. Chapter 4 gives an overview about the state of the art involving formal verification for kernels. Chapter 5 contains the development of the work, a formal model of the secure partitioning microkernel followed the CbyC approach. Chapter 6 concludes this dissertation by describing the most relevant conclusions of the work herein described.
Chapter 2

High assurance software development with SPARK

In this chapter formal methods are reviewed. We separate them in three distinct categories: the classical approaches, the lightweight approaches, and the approaches more directed to the code. We also introduce the SPARK language and the development methodology that was used in this work.

2.1 Formal methods

Formal methods (FM) involve the application of mathematical techniques, most often supported by development tools. They can be applied in different situations with system models that specify every detail of the implementation or only the most abstract requirement, with different notations and different tools. FM are used in software specification to obtain a precise statement of what the software does (its functionality), not concerned about how to do it. They are also used in the implementation with the propose of code verification.

Non safety-critical systems, normally use an informal method via some combination of testing and inspections to verify the software with respect to functional requirements. Failures that are not detected in these informal inspections are found later, with the use of the software itself and with a huge cost associated. When speaking of software
for critical systems, this kind of informal verification is inadequate and more rigorous techniques are necessary. In some cases, the regulatory agencies are the ones that require more rigorous methods, such as the use of FM.

There is a big difference in attitude with respect to FM. Generally speaking, FM are seen as a key in academia; on the other hand, are seen as somewhat irrelevant in the industry, with the exception of critical systems industry. Although there are some changes with respect to greater acceptance and use of FM, there is still a long way to go to achieve a general dissemination of their use in other areas beyond the critical areas.

Nevertheless there is a change of mentality after the first utilization of FM, as shown in Table 2.1. The survey summarized in this table also shows that in the projects under study the use of FM has always been of great success and in 75% of cases was even assumed the recommendation of a similar techniques in new projects.

<table>
<thead>
<tr>
<th></th>
<th>Improvement</th>
<th>Worsening</th>
<th>No effect/no data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>35%</td>
<td>12%</td>
<td>53%</td>
</tr>
<tr>
<td>Cost</td>
<td>37%</td>
<td>7%</td>
<td>56%</td>
</tr>
<tr>
<td>Quality</td>
<td>92%</td>
<td>0%</td>
<td>8%</td>
</tr>
</tbody>
</table>

Table 2.1: Time, cost and quality with use of Formal Methods - from [9].

An old adage of FM says that only proofs can show the absence of bugs, tests on the systems can only show their presence. Unlike tests that usually can only cover a small subset of possible executions, FM have the ability to cover all of these executions. Yet we must bear in mind that a proved software only gives us confidence about the theorems, we need to trust that the formalization was well done in the first place. The software can be well coded, without errors in its processing, and yet it may well not do what it was developed for. Do not just build a software and be confident that it works as expected. After building a software convincing the others that it is indeed a good product is necessary. To help in this process, validation is used, which consists of software tests in order to show that it really does what is supposed to.

Currently, formal methods are divided into two main verification techniques: theorem proving and model checking.
The theorem proving approach consists of the description of the desired system properties in a formal logic, with a set of axioms, pre- and post-conditions in order to build a model. Secondly, a mathematical proof is conceived with the aim to ensure that the model satisfies the desired properties. Although theorem provers have evolved significantly, by removing some of the tedious processes of the proof, they still need human intervention to guide the more complex proofs. This is a process that requires significant knowledge and is seen as the disadvantage of theorem proving.

One way in order to facilitate the process, with the aim to automatically discharge more proofs, can be achieved with tools that provide the best possible solution to discharge the proof - as with in Frama-C [10], where the proof can be discharged in several different provers in a manner that benefit from the different characteristics of each prover. This allows that different parts of the program are proved with different provers.

On contrary, the model checking has a greater automatism, thus requiring less human intervention. Typically, model checking works on a model that contains only the necessary and relevant system properties. This is necessary because the accessible state space of a model will be thoroughly explored to determine if the properties are maintained; obviously, this space should not grow in an unnecessary manner. The size of state space thus becomes a handicap in contrast to what happens in theorem proving where the state space is not a problem - and indeed makes it effective in more complex systems and in full functional correctness. Typically, in these systems with more complexity, the model checking is only used to verify some specific properties, not the whole system. Besides the verifications on a model, there are tools like SLAM [11] that work directly on the code, but once again, the properties checked are simple and do not cover the whole system. Unlike theorem provers, the model checking is more accessible for relatively inexpert users. There have been major advances with regards to the state space explosion problem, which makes the model checkers a very practical tool especially if placed in a native way in the IDE commonly used by developers.

Some FM are presented below divided into three distinct classes: the classical approaches, the lightweight approaches, and the approaches more directed to the code. The classical and the more directed to the code approaches comprise methods with theorem proving but provide simultaneously model checking. The lightweight approaches comprise only the model checking method.
2.1.1 Classical

This classification of classical FM refers to the traditional approaches, which invokes set theory and first order logic. Therefore, the three main model-oriented FM [12] are presented below: the Z notation, the B-Method and the VDM. These are also the most popular formal software development methods [13].

- The Z (pronounced "zed") notation is a specification language based on set theory and first order predicate calculus. It has been developed in the late 1970s, by the Programming Research Group (PRG) at the Oxford University Computing Laboratory (OUCL), inspired by a Jean-Raymond Abrial work. The notation was defined formally by Spivey in 1988 [14]. In order to facilitate the specification process, Z notation allows the decomposing of the system specification into smaller components. The individual components are then combined at the end in order to describe the entire system as a whole. Z models are usually accompanied by a narrative in a common language that helps readers, writers and reviewers to understand the models. An introduction to this notation can be found below in section 2.3.1.1 - Formal specification.

The notation has a wide range of support tools\(^1\), some of them are described below:

- The Community Z Tools (CZT) project [15] is an open source project providing an integrated toolset to support Z notation;
- Fuzz [17] is a type checker created by Spivey for the Z language;
- ProofPower [18] is a suite of tools supporting specification and proof in Higher Order Logic (HOL) and in the Z notation;
- ProZ [19] is an extension of the B-Method tool ProB, that offers some support for model checking and for animating Z specifications;
- HOL-Z [20] is an interactive theorem prover for Z based on Isabelle/HOL.

- The B-Method [21] is a formal development methodology based on set theory with first-order logic. It uses the abstract machine notation (AMN) as specification language and allows progress from an initial high-level specification all the way to the implementation via formal refinement. B-Method has been developed by Jean-Raymond Abrial (the originator of Z notation). B-method has been successfully

\(^1\)The reader is referred to the section on Z notation available from the Wiki, set up by Jonathan Bowen [15], were a more complete list of tools can be found.
used in the development of many complex high integrity systems, as the driverless
metro line 14 in Paris \[22\] or the driverless shuttle for Paris-Roissy airport \[23\].

The main tools of B-Method are:

– The B-Toolkit \[24\], which is a set of integrated tools which fully supports the
  B-Method for formal software development;

– Atelier B \[25\], which is the more complete tool for B-Method, include features
  like type and static semantics checking, proof support and refinement;

– ProB \[19\], which is an animator and model checker for B-Method;

– The Rodin Platform \[26\] is an open source Eclipse-based IDE for Event-B
  language. Event-B is an evolution of B-Method and one of the main reasons
  for this evolution was simplicity \[27\].

• The Vienna Development Method (VDM) is a complete software development
  method, whose features are modeling computing systems, analyzing models and
  progressing to detailed design and coding. It has been developed in the IBM
  Vienna Laboratory in the mid-1970s. VDM is a method which uses a specification
  notation that is similar to Z. The first complete exposition of the notation and
  method was made in \[28\]. The original VDM Specification Language (VDM-SL)
  \[29\] was extended for VDM++ \[30\]. The main difference between VDM-SL and
  VDM++ notations are the way in which structuring is dealt with. In VDM-SL
  there is a conventional modular extension, whereas VDM++ has a traditional
  object-oriented structuring mechanism with classes and multiple inheritance.

VDM has two main sets of support tools. They are:

– The VDMTools \[31\] is the leading commercial tool for VDM-SL and VDM++.
  It supports syntax and type checking, includes a test coverage tool and also
  includes automatic code generation;

– Overture \[32\] is an open source project that has the objective of developing
  the next generation of open-source tools for VDM. In addition to support
  VDM-SL and VDM++, these tools also offer support to VDM-RT (the new
  extension of VDM that is concerned with real-time and distributed systems).
  The Overture tools are written entirely in Java and build on top of the Eclipse
  platform.
2.1.2 Lightweight

Lightweight formal methods is a recent approach that combines the classical ideas of a formal specification from methods like those presented above, and verification with new automatic checking processes, usually for partial problems. They have the capacity to yield results with a fast and easy application. They are usually applied with more modest goals and the used tools require less specific knowledge to apply them. They can be seen as a way of facilitating the incursion of the FM in the industry in general, because they can be applied/added in the development cycle typically used in the company.

There are many works that use “heavyweight” methods in a light manner \cite{33-36}, i.e., they only implement a piece of technology/methodology, or apply it to a single part of the system. We do not present these methods in this section as they appear in sections 2.1.1 and 2.1.3. We give a special emphasis to model checking methods because their use is often underestimated and people sometimes not perceive the potential and benefits over standard tests. The model checker finds bugs in a more reliable way than normal tests, since it verifies a state space with a much higher size.

Below are presented some of the lightweight formal methods commonly used:

- Alloy is a lightweight modeling language for software design. Alloy was developed at MIT by the Software Design Group under the guidance of Daniel Jackson \cite{37}. The language is strongly influenced by the Z specification language presented above. The support tool is the Alloy Analyzer that provides a fully automatic analysis and provides a visualizer that shows the solutions and counterexamples that it finds.

- SPIN \cite{38} is a popular model checker developed in 1980 at Bell Labs in the original Unix group of the Computing Sciences Research Center. The formal models are written in PROMELA (Process Meta Language). SPIN verifies the correctness of distributed software models in a rigorous and mostly automated fashion and provides support for linear temporal logic.

- UPPAAL \cite{39} is a model checker developed in collaboration between the Department of Information Technology at Uppsala University in Sweden, and the Department of Computer Science at Aalborg University in Denmark. This tool is indicated for real-time systems modeled as networks of timed automata. It enables three model development stages: modeling, simulation/validation, and verification.
• TLC \cite{talc} is a model checker and a simulator for specifications written in TLA+
\cite{tal}. TLA+ is a specification language based on TLA (Temporal Logic of Actions)
\cite{tlal} and was developed by the Microsoft research center. It is particularly useful
for describing concurrent and distributed systems. The TLC model checker is
included in the main tool for TLA+, the TLA Toolbox \cite{taltoolbox}.

2.1.3 Directed to the code

This section focuses on the methods directed to the source code. In these methods, as
first suggested by Hoare logic \cite{hoarelogic}, assertions are used in the program code. The design-
by-contract\footnote{Design-by-Contract is a trademark of Interactive Software Engineering Inc.} is one of the code oriented formal methodologies. It receives highlight in
this section because part of this work uses the methodology.

The term Design by Contract was originally introduced by Bertrand Meyer \cite{meyer}. He was
the founder of Eiffel \cite{eiffel1,eiffel2}, the first language that supports the paradigm. Beyond
Eiffel, there are other languages that support also Design by Contract such as Spec# \cite{spec#},
SPARK \cite{spark} (which will be used in this work), C with ACSL\footnote{ANSI/ISO C Specification Language.} and tools like frama-C
\cite{frama-c}, Java with Java Modeling Language(JML) \cite{jml} or ESC/Java2 \cite{escjava2} and many more.

As defined in \cite{spaser}, a contract is a technique for specifying the obligations of participat-
ing objects. It is intended to formalize the collaboration and behavioral relationships
between objects in a non-ambiguous manner. A contract therefore defines communicat-
ing participants and their contractual obligations. The contractual obligations not only
comprise traditional type signatures but also capture behavioral dependencies. With
the help of formal verification methods, it is possible to prove that software behavior is
correct in respect to these specifications.

The contracts must be written in a previous stage than that of the code since they are
more abstract. This gives us a first model of the application design. The improvement
of this first approach is accomplished by refining the contracts. We are only ready
to move on to the code implementation when the architecture is stable. Contracts
enable the storage of details and assumptions towards the documentation of software
components and API usage. It avoids for example the constant check of arguments in
operations. Moreover, contracts are also important during the coding process to inform
the programmer what he/she is supposed to do.
Chapter 2. *High assurance software development with SPARK*

The usefulness of contracts depends naturally on their quality and pertinence. In the next sections we will try to show how this can be achieved.

### 2.2 SPARK overview

SPARK consists of a high level programming language. It has been conceived for writing software for high integrity applications, where it is mandatory that the program is well written (without errors). High integrity applications include both safety and security. Safety critical applications are usually defined to be those where if program is in error, life or the environment are at risk, whereas security applications concern the integrity of information or the access to it. Of course, any application benefits if the program is well written from the beginning and SPARK enables the prevention of errors since the first stages.

In a safety-critical real-time system, a “run-time error” can be quite as hazardous as any other kind of malfunction: all language violations must be detected prior to program execution. This was taken into account in the design of SPARK.

SPARK can be used at various levels. Data flow analysis is the simplest level. This analysis enables to find errors like mistaken identity or undefined values that are not used. The intermediate level of information flow analysis enables control with respect to inter-dependence between variables. Finally, the major analysis level enables formal proof. This kind of analysis is used in applications with high requirements of integrity. This analysis uses formal pre-conditions, post-conditions and other assertions. One program can use various levels of analysis at the same time. For the most important part formal analysis can be used, and for the rest of the program a more weak analysis.

SPARK is sometimes regarded as being just a subset of Ada with various annotations that you have to write as Ada comments. This is true, but nevertheless, in accordance with John Barnes [53], SPARK should be seen as a distinct language. It is justified because SPARK contains the features required for writing reliable software and enables techniques for analysis and proof according to the requirements of the program.

SPARK shares compiler technology with the standard language Ada, and this seems the perfect choice, since Ada has a good lexical support for the concept of programming by contract. Ada enables the paradigm programming by contract (already seen in previous section) in a natural way. Ada permits a description of a software interface (the
contract) independently from its implementation (the code). They can be analyzed and compiled separately because Ada contains a structure separating interface (know as a specification) from the implementation (a body). This applies to individual subprograms (procedures or functions) and to groups of entities encapsulated into packages. This is one reason why Ada was chosen as base for SPARK.

In order to allow more information on the behavior of the implementation, annotations are added to the specification and body. These annotations have the form of Ada comments. There is no need for “strange” and “complicated” annotations, only simple statement about access permissions that allow to verify if there are any inconsistencies between what is pretended (the annotations) and what is really being done (the code).

The annotations can be divided in two categories, the first concerns flow analysis and visibility control and the second concerns formal proof.

Getting strong contracts (accurate and relevant annotations) is a major objective of SPARK, so as to move all the errors to a lower category or ideally find them all before running the program.

Indeed, it is clear that SPARK is not a subset of Ada at all since SPARK imposes additional requirements through the annotations. A SPARK program can be compiled by a standard Ada compiler, because the annotation are Ada comments, and will be ignored by the compiler.

The relationship between Ada and SPARK can be seen in Figure 2.1; the overlap between them refers to the kernel. The kernel does not contemplate the exceptions, generics (templates), access (pointers) types, or goto statements, because they create difficulties in proving that a program is correct.

The SPARK core annotations (flow analysis and visibility control) are divided into several types. There exist two important annotations used to increment information given by the normal Ada specification: (i) the global definitions, declare the use of global variables by subprograms - is called data flow analysis and only comprise the direction of data flow (only use the global annotation); and (ii) dependency relations of procedures, specify the information flow between their imports and exports via both parameters and global variables - is called information flow analysis and uses in addition the dependency between variables (using the derives annotation). The annotations for access variables in packages are also very important. They allow modularity between components. The
encapsulation in the sense of Object Oriented Programming (OOP) is obtained by packages. The packages control the access to hidden entities via subprograms (methods). There are three types of annotations related with packages: (i) the inherit clauses controls the visibility of packages, (ii) the own variable clauses controls access to variables of packages, and (iii) the initialization annotations imposes the initialization of own variables, avoiding some common errors. The own variables describe the state of packages. They can be used to represent values in the physical world. As mentioned earlier, all of these annotations should be written in a early stage of design, before coding stage.

For a better understanding of the annotations described above, we present an example. In this case a simple example of a stack, a last in, first out (LIFO) structure, but which can express the several types of annotations that can be used.

First, we have the specification of the stack, which is shown below in Code Listing 2.1.

In this specification we can see the use of own clause at line 2, thus ensuring that the package has a variable called “State”. On the next line we see the initializes clause, it will require that the “State” variable is initialized in the body of the package. Later, we have the specifications of “Push” and “Pop”, the operations that can be made by the stack. For the “Push” procedure, which starts in line 5, we see, as expected, that one value is passed on as parameter. This value represents the value entered in the stack, it is of type “in” because it will be inputed. In lines 6 and 7, we have the annotations of “Push” procedure, first the global clause that expresses how the procedure uses the variable “State”, it has “in” and “out” modes, which means that the procedure will read
and update the variable in the procedure. In the next line, we have the *derives clause* that tracks dependencies, in this case we can observe that the final value of the variable “State” depends not only on its initial value, but also depends of the value of the input parameter “X”. A similar process is done for the “Pop” procedure, with the difference that the parameter “X” is of output type. The “X” represents the value that was on top of the stack. Another difference is in line 11, in the *derives clause*, where, both the final value of the variable “State” and the value of the parameter “X” depend on the initial value of variable “State”.

---

```platon
package TheStack

--# own State;
--# initializes State;

is

procedure Push(X : in Integer);
  --# global in out State;
  --# derives State from State, X;

procedure Pop(X : out Integer);
  --# global in out State;
  --# derives State, X from State;

end TheStack;
```

---

**Code Listing 2.1:** Stack specification package

After the completion of the specification, we show in Code Listing 2.2 the package body that will implement the previous specification. In this case, we can see that in line 2 we have again an *own clause*, but in this case represents a refinement, the variable previously known simply as “State” is now transformed into two variables, the variable “S” and the variable “Pointer”. Between lines 4 and 9 we declare the types. We can see that variable “S” is a vector of integers and its indexes are of a range between 1 and 100, and the variable “Pointer” is of “Pointer_Range” type, which means that its value is between 0 and 100.

If we had not refined the variable “State”, annotations on procedure bodies would not be needed, but, as we insert detail in variable “State”, we also need to detail the operating procedures with the new annotations.

For the “Push” procedure, we can see in line 12 that variables “S” and “Pointer” are both “in” and “out” modes, i.e. are input and output variables. In the next line, we have the *derives clause* for variable “S”, which describes that the value of variable “S” not only depends on its own initial value, but also on both the value of “X” parameter.
and the value of the “Pointer” variable - this was expected because the value of “X” will be entered into the vector position corresponding to the value contained in “Pointer” variable. In line 14, we have the continuation of the derives clause, in this case for “Pointer” variable, that says the final value of the “Pointer” depends on its initial value. In lines 17 and 18, we can see the behavior of the program and confirm that the previous clauses are in compliance. Thus, we have that the final value of the variable “Pointer” is incremented by one unit and that the “S” vector undergoes an update on the position with the value contained in the “Pointer” variable, with the value that was passed on in “X” parameter.

```
package body The_Stack
  --# own State is S, Pointer; -- refinement definition
is
  Stack_Size : constant := 100;
  type Pointer_Range is range 0 .. Stack_Size;
  subtype Index_Range is Pointer_Range range 1 .. Stack_Size;
  type Vector is array(Index_Range) of Integer;
  S : Vector;
  Pointer : Pointer_Range;

procedure Push(X : in Integer)
  --# global in out S, Pointer;
  --# derives S from S, Pointer, X &
  --# Pointer from Pointer;
  is
  begin
    Pointer := Pointer + 1;
    S(Pointer) := X;
  end Push;

procedure Pop(X : out Integer)
  --# global in S; in out Pointer;
  --# derives Pointer from Pointer &
  --# X from S, Pointer;
  is
  begin
    X := S(Pointer);
    Pointer := Pointer - 1;
  end Pop;

begin  -- initialization
  Pointer := 0;
  S := Vector‘(Index_Range => 0);
end The_Stack;
```

Code Listing 2.2: Stack body package
Chapter 2. High assurance software development with SPARK

Starting at line 21, we have the body of “Pop” procedure. In this case, we see that “Pointer” variable continues to be of “in” and “out” modes, input and output respectively - as happened in the specification of “State” variable - but “S” variable is only of “in” mode, or just input - this is due to the fact that “S” is not updated in this procedure. In line 23, begins the derives clause and we observe that the final value of the “Pointer” variable depends only on its initial value. In the next line, we see that the value of the parameter “X” depends on the values of “S” and “Pointer” variables, this its because “X” will take the value contained in the vector “S” at the position equals to the value contained in “Pointer” variable. In lines 27 and 28, we can confirm that the operation does what expected, i.e. the “X” takes the value of the vector “S” in the position of the value contained in “Pointer” and the “Pointer” is decremented by one unit, depending its final value only on its initial value, without depending on any other variable.

Last but not least, we have the initialization of “Pointer” and “S” variables, thereby complying with initializes clause it was put in the specification of the package.

To ensure that a program cannot have certain errors related to the flow of information (such as, the use of uninitialized variables and the overwriting of values before they are used), the Examiner needs the SPARK language with its core annotations. However, this kind of annotations does not address effectively the issue of dynamic behavior. To deal with this, some proof annotations are added to allow analysis of dynamic behavior prior to execution. The proof annotations can be as follows:

- Pre- and post-conditions of subprograms;
- Assertions, such as loop invariants and type assertions;
- Declarations of proof functions and proof types.

Continuing with the previous example of the stack, we show in Code Listing 2.3 how proof annotations on “Push” and “Pop” procedures can be added. These annotations should have been written before the code by another person other than the one who wrote the code. For the “Push” procedure, a pre-condition annotation that checks whether the value of “Pointer” is less than the value of “Stack_Size” is in line 39 - this is to prevent to access a position outside the bounds of the array. In the following two lines the post-condition annotation was included - that shows us which values of “S” and “Pointer” variables are obtained after the execution of the procedure. The final value of
“Pointer” is equal to its initial value (“Pointer~”) increased by 1 unit, as exactly as in the body of the operation. The same happens with “S” that will update the position of the value contained in “Pointer” with the value of “X”.

For the “Pop” procedure, a pre-condition annotation is entered in line 52 - that checks whether the value contained in the “Pointer” is not zero, i.e. checks if the stack is not empty; otherwise “Pop” can not run, a “Push” must be performed before that. In lines 53 and 54 we have the post-conditions annotations which again are in accordance with the code. They mean that the final value of “Pointer” is equal to its initial value decremented by one unit, and that the parameter “X” takes the value of “S” at the index position contained in “Pointer”.

```spark
procedure Push(X : in Integer)
--# global in out S, Pointer;
--# derives S from S, Pointer, X &
--# Pointer from Pointer;
--# pre Pointer < Stack_Size;
--# post Pointer = Pointer~ + 1 and
--# S = S~[Pointer => X];

is
begin
  Pointer := Pointer + 1;
  S(Pointer) := X;
end Push;

procedure Pop(X : out Integer)
--# global in S; in out Pointer;
--# derives Pointer from Pointer &
--# X from S, Pointer;
--# pre Pointer /= 0;
--# post Pointer = Pointer~ - 1 and
--# X = S(Pointer~);

is
begin
  X := S(Pointer);
  Pointer := Pointer - 1;
end Pop;
```

Code Listing 2.3: Push and Pop procedures with proof annotations

With this example we are able to show the various types of annotations and how they are used. The annotations are quite affordable and they allow the use of Examiner to check the code (program).
Technically, the SPARK was born through the work of Bob Phillips in 1970 at the Royal Signals and Radar Establishment. The study consisted of understanding and analyzing the behavior of existing programs and developed tools to perform such analysis. The idea gained notoriety with the importance of the correction of software for applications with critical security. In 1994 there appears the first version of SPARK (based on Ada 83) produced at the University of Southampton (with the support of the Ministry of Defense of the United Kingdom) by Bernard Carré and Trevor Jennings.

Later, the language was gradually augmented and refined, first by Program Validation Limited and later by Praxis Critical Systems Limited. In 1995 the Ada language was revised, which resulted in the Ada 95, and in 2002 SPARK was adjusted so the language corresponded to the version of Ada 95. In 2004, Praxis Critical Systems Limited changed its name to Praxis High Integrity Systems Limited and in 2007 appears a new version of the standard, called Ada 2005 to be distinguished from the previous version. More recently, in 2009, a partnership with AdaCore resulted in the release of “SPARK Pro” under the terms of GPL. In the middle of 2009, the SPARK GPL 2010 Edition emerged with an initial set of new features for SPARK 2005. In 2010 the company merged with SC2 to form Altran Praxis, which is now the company responsible for maintaining and developing SPARK. More details on the history of SPARK can be found at [3, 53].

Although SPARK has emerged from a study where the initial objective was to verify existing programs, the primary objective now is to write programs correctly.

### 2.3 Development methodology

In all engineering disciplines it is widely accepted that it is better to avoid the introduction of errors beforehand rather than having to correct them at a later stage. For example, as far as a manufactured physical component is concerned, realizing at a late state that an entire batch of such a component was made out of the specifications, has huge costs. This may also happen in software development. Generally, the later the errors are found, the more expensive is to eliminate them. Those corrections may represent a very large percentage of the development costs when we talk about critical systems with high standards. The only way of obtaining the high levels of integrity required for critical software at an acceptable cost, is by pursuing development methods that make it hard to introduce errors and which facilitates their early detection. This is achieved by an approach sometimes termed as Correctness by Construction [3].
2.3.1 Correctness by construction

This process has been successfully used by Praxis in many projects. The proposed process by Altran Praxis consists overall of the following five steps:

1. Requirements analysis;
2. Formal specification (using the formal language Z);
3. Design (INFORMED process);
4. Implementation in SPARK;
5. Verification (using the SPARK Examiner toolset).

As expected, requirements analysis is essential in any software project, so we need to make explicit, first, what the software is supposed to do, and secondly, what their features and peculiarities are. However, the passage of the requirements in text form to implementation in code is not always easy to do, mostly because they have not specified in an unambiguous way how to be implemented. To circumvent this problem, it is recommended the modeling of requirements in order to eliminate possible ambiguities in the requirements and for better understanding of the problem. One of the modeling languages mostly used for this purpose is the Unified Modeling Language (UML), which allows (depending on the type of the UML models) a more visual perspective of the requirements and the software functionalities. There is an extensive list of UML model types, and its creation does not always follow the same pattern, making it unclear and informal, allowing more than one interpretation with the same model or introduces difficulties to disambiguate certain requirements. On the other hand, using a formal modeling language, it is possible to express the requirements in an unambiguous way, so that there is only one interpretation of the problem.

The SPARK language and its support tool, the Examiner, are designed expressly to support the CbyC paradigm, allowing an early stage detection of errors. This avoids the normal validation by testing at the end of a project, with all the costs associated. A key ingredient of the CbyC approach is an effective design method known by INFORMED. This method is well defined by Praxis. Below we present its major aspects. This method builds on the minimization of the flow of information between subsystems. INFORMED is an INformation Flow ORiented MEthod of (object) Design. Unnecessary flow of information between different parts of the system increases considerably the complexity
of the SPARK annotations and consequently the difficulty of proofs. This can be done by minimizing propagation of unnecessary detail with: a) a correct localization and encapsulation of state; and b) avoiding making copies of data with an appropriate use of hierarchy. Another principle is a clear separation of the essential from the inessential. This gives priority to the core functionality of the system.

In the next two sections, we present steps 2 and 3 of the development process, the Formal specification step (which in this case is made with the formal language Z) and the Design step (that consists in the use of INFORMED process) respectively.

### 2.3.1.1 Formal specification

As mentioned earlier, it is extremely important to create a formal model of the problem. This methodology uses the Z notation to achieve this.

Hereby we introduce the basics of Z notation, with its types and schemas.

The main building block in Z is a **schema**. A Z **schema** takes the form of a number of state components and, optionally, constraints on the state.

```
SchemaName
  declarations
  constraints
```

The next schema represents a clock with hours and minutes. The obvious two constraints are introduced: a) the value of the hours (h) must be between 0 and 23; and b) the value of the minutes (m) must be between 0 and 59.

```
Clock
  h : N
  m : N

  h < 24 ∧ m < 60
```

Schemas are used to describe behavior under change. To represent the new values of the state, the “′” single quote is used.
We can describe a simple increment in a minute of our clock. Note that the constraints introduced in the above schema are still valid.

\[
\begin{align*}
\text{IncrementMinute} & \\
\Delta \text{Clock} & \\
\;
\end{align*}
\]

\[
\begin{align*}
h' &= h \\
m' &= m + 1
\end{align*}
\]

The \( \Delta \text{Clock} \) means that a change occurs on the state. Another useful definition is \( \Xi \text{Clock} \) which describes the case where the state of the schema is unchanged. For example, for operations that only have “read” permission.

In addition to schemas, \( Z \) allows us to define basic types which will be used thereafter in the components of our schemas. Next, we introduce the concept of “alarm”, that can be used to extend a simple clock to an alarm clock.

\[
[\text{ALARM}]
\]

We can introduce axiomatic descriptions like:

\[
\begin{align*}
\text{seconds} & : \mathbb{N} \\
\text{seconds} & \leq 59
\end{align*}
\]

We introduce the seconds with an obvious constraint, that they must be contained between 0 and 59, and then let us introduce the alarm clock, with hours and minutes of the previous Clock plus seconds and a buzz.

\[
\begin{align*}
\text{AlarmClock} & \\
\text{Clock} & \\
\; \\
\text{seconds} & : \text{seconds} \\
\text{buzz} & : \text{ALARM}
\end{align*}
\]

Only the basic notions of language were presented, with its basic elements. For a more complete description of the language please read [54].
2.3.1.2 INFORMED

The INFORMED design approach is a very important step because describes how SPARK deals with properties of good software design, such as:

- **Abstraction**: this enables us to ignore certain levels of detail. Hiding unnecessary detail allows us to focus on the essential properties of an object;

- **Encapsulation**: it is a separation of specification from implementation. The users of an object should not be concerned with its internal behavior;

- **Loose coupling**: coupling is a measure of the strength of connections between objects. High levels of coupling make modifications difficult to be performed as changes occur in more than one place. On contrary, loose coupling makes modifications easy to occur;

- **Low Cohesion**: cohesion is a measure of the strength of connections between an object attributes. A low degree of separation of attributes (low cohesion) allows us to perform changes in a single attribute leaving the remaining unchanged. For example, a car has an engine and doors, which represent two distinct attributes of the car object; any change in the driver’s door does not affect at all the engine of the car.

- **Hierarchy**: certain objects are contained inside others and cannot be reached directly. For example, when we see a car we see immediately the doors too, but to see the engine is necessary to access the car and open the hood.

The INFORMED is composed by five blocks, that form the basic design elements:

- **Main program**: this control the behavior of the entire system. It requires the SPARK annotation \texttt{main\_program};

- **Variable package**: this is the same as usually known as abstract state machine or an object for example in Java. It is a SPARK package that contains static data or “state”. It is annotated by an \texttt{own} variable annotation;

- **Type package**: known as abstract data type or a more vulgar class. It is also a SPARK package although in this case does not have state;

- **Utility Layer**: introduces operations to the system but does not introduce state or types;
• **Boundary Variable**: these variables are used to represent real world entities.

The following notation, shown in Figure 2.2, is used to express the diagrams of the system structure:

![INFORMED notation diagram](image)

**Figure 2.2**: The INFORMED notation - image adapted from [55].

The arrows represent the dependence between elements (the element at the arrow head is used (inherited) by the element at the arrow's tail) and can be separated into:

- **Strong coupling**: represents use, either directly or indirectly of the global variables of a package;
- **Weak coupling**: represents use of types and utilities provided by a package without using or affecting the state of the package.

The INFORMED design normally follows a few steps. The process consists of finding a solution for each step and test them with the goal to move on to the next step. If one step is becoming too complicated, possibly a wrong choice has been made in the previous step. The steps are:

1. **Identification of the system boundary, inputs and outputs**
   First, choosing and delineating the boundary system. Also identification and selection of the physical inputs and outputs, as described in Figure 2.3.

2. **Identification of the SPARK boundary**
   The selection of the boundary variables defines the SPARK system boundary, this can be seen in Figure 2.4. The abstracted input/output values provided by the boundary variables are the Input Data Items and Output Data Items from Figure 2.3.
3. Identification and localization of system state

Useful systems store data values in variables and therefore have “history” or “state”. Selecting appropriate locations for this state is probably the single most important design decision that influences the amount of information flow in the system. The decisions involve deciding what must be stored, where it must be stored and how it should be stored.

4. Handling initialization of state

State variables can be initialized using two distinct and separate approaches: a) during program elaboration (the variable is considered to have a valid value prior to execution of the main program); and b) during execution of the main program by a program statement.

5. Handling secondary requirements
Software designers frequently have to reconcile conflicting requirements; Amongst these requirements are some which INFORMED describes as secondary “requirements” because, although they may be important to the success of the project, they are not derived from the core functionality of the system being designed.

6. Implementing the internal behavior of components

Initially only annotated specifications for these objects are required allowing early static analysis of the design. The first step should always be to see whether decomposition into further, smaller INFORMED components is possible.

2.3.2 Tools

The tools are one of the strengths of the methodology, since they are in constant development. Apart from being a complete package, it starts with a simple type check until it may reach the proofs.

The SPARK tools are in an advanced phase of maturation, and may even be included in the environment development GNAT Programming Studio – IDE.

The key SPARK tool is the Examiner [56]. It has two basic functions:

- It checks conformance of the code to the rules of the kernel language;
- It checks consistency between the code and the embedded annotations by controlling data and information flow analysis.

There is a high degree of confidence in the Examiner, once it was written in SPARK and was submitted to itself [53]. The Examiner performs two kinds of static analysis. The first, made up of language conformance checks and flow analysis, checks that the program is “well-formed” and is consistent with the design information included in its annotations (this analysis englobes the two basic functions listed in the preceding paragraph). This stage is extremely straightforward and can be readily incorporated into the coding phase of the development process. After this, it checks if the source code is free of erroneous behavior and free of conditional and unconditional data flow error. The second (optional) kind of analysis is verification, which shows by proof whether the SPARK program has certain specified properties. The most straightforward is a proof that the code is exception free, this adds Constraint_Error to the list of possible errors eliminated by SPARK. Proof can be used to demonstrate, unequivocally, that
the code maintains important safety or security properties or even to show its complete conformance with some suitable specification.

Based on proof annotations, the Examiner generate verification conditions (potential theorems) which then have to be proved in order to verify whether the program is correct with respect to the annotations. The verification conditions can be proved manually in a process usually tedious and unreliable, or by using other tools such as the Simplifier [57] and the Proof Checker [58].

The Simplifier is an automated theorem prover that processes the verification conditions (VCs) produced by the Examiner. The proof of these VCs confirms critical program properties, such as the freedom from run-time errors and exceptions, or specific safety and security properties. The Simplifier main purpose is to simplify verification conditions prior to developing a proof, but in many cases is able to reduce all the conclusions to True. Any remaining undischarged conditions may be proved with the assistance of the Proof Checker. This is an interactive proof tool that uses the same logic and inference engine as the Simplifier. For a more readable visualization of results POGS (Proof Obligation Summary Tool) [59] summarizes the semantic output files produced by the Examiner, the Simplifier and the Proof Checker.

The use of tools like Examiner encourages the early use of a V&V (Verification and Validation) approach. This is made possible with code written in SPARK with appropriate annotations and that are now able to be processed by Examiner, even if it still can not be compiled. This is clearly the recommended approach, strongly discouraging the consideration of an existing piece of Ada code and then add to it the annotations (known as “Sparking Ada”). This is because it typically leads to extensive annotations indicative of an unnecessarily complex structure. To avoid this, the annotations should be seen as part of the design process.

\section{2.4 Summary}

The use of formal methods is important and even necessary for certain types of software. Its use leads to more robust and more reliable software. There are various types of FM that can and should be used for distinct purposes. As suggested by the development method CbyC, SPARK needs to be introduced in the beginning of the software development, not only in the common validation and verification stage, thus including also this phase of the process from the very beginning. The INFORMED approach capture
system design information in annotations and use it to influence the shape and characteristics of the software implementing that system. The use of SPARK at the design stage of the life cycle facilitates a cost-efficient CbyC development method.
Chapter 3

Case study: a secure partitioning microkernel

The use of embedded devices has significantly grown in the last years. We have this type of devices in any place around us. In cars, in mobiles, in houses and much more. This type of devices is increasingly connected to the Internet. This can be a big vulnerability because it creates opportunities for malicious users to exploit security weaknesses and take control of these systems. These malicious people can access confidential information, disable a critical system, or modify its default behavior. It is therefore necessary to develop highly secure embedded systems to ensure our safety. Besides those already mentioned, these embedded devices are also located in critical areas for homeland security, such as Internet service providers, financial institutions or power companies. For this, it is urgent to ensure the software of these devices is highly secure. In particular, it is necessary that the embedded operating system is secure or “bullet proof”. The OS is vital, because any application or security mechanism implemented could be bypassed, if the OS is not safe in the first place.

The development of a commercial operating system usually follows the “Penetrate and Patch” approach [60]. OS are attacked by viruses, worms and trojan horses, highlighting their vulnerabilities. Thus vendors developed and released patches to fix the vulnerabilities, almost by a trial and error method. Clearly this kind of approach is ineffective for the systems we are talking about. The right solution is to provide an approach where systems are designed to be safe and secure from the very beginning. To meet this need,
the certification of software comes up, which allows a gain in confidence on the software according to the certification level of it.

Various standards and criteria for certification of software have been created. Common Criteria [61] is one of the most important. It is an internationally accepted standard to specify and evaluate security assurance. The evaluation of security software through the Common Criteria standard defines Evaluation Assurance Levels (EAL 1-7) that indicates the process rigor associated with the development of an information technology product, as shown below:

- EAL-1: Functionally tested;
- EAL-2: Structurally tested;
- EAL-3: Methodically tested and checked;
- EAL-4: Methodically designed, tested, and reviewed;
- EAL-5: Semi formally designed and tested;
- EAL-6: Semi formally verified, designed, and tested;
- EAL-7: Formally verified, designed, and tested.

Assurance levels start at EAL-1, the lowest level, and increases until EAL-7, the highest level. Nowadays, the EAL-5 is considered the minimum acceptable level for critical systems. And the search for EAL-7 is constant, in order to ensure high reliability of the systems. The EAL-7 involves formal verification of the software product using mathematical models and theorem proving.

As we have seen, the OS (or in more basic way and more low level, simply kernel), is one of the most important parts of the system, and must be safe and secure in first place, so that other applications (even these certified to high levels of assurance) are not subject to vulnerabilities. However, due to their size and complexity, a kernel is difficult to certify. In this context the concept of microkernel arises.

In the next section, we introduce the notion of microkernel which, as the name suggests, is a kernel of tiny dimensions that contains only the essential features. In section 3.2 we introduce the notion of separation kernel and its benefits.
A microkernel is quite different from a conventional (monolithic) kernel, as shown in Figure 3.1. In the conventional kernel, all OS services run with kernel permissions and reside in the same memory area. Contrarily, in the microkernel all things that might potentially bring down the system run outside of kernelspace, in individual processes often known as servers. This enables that, if something goes wrong, just the faulty component needs to be restarted. In this way, it does not crash the hole system and there is no down time. Another advantage of a microkernel is its simplicity. In a microkernel, each driver, filesystem, function, etc., is a separate process located in userspace. This means that a microkernel is relatively simple and easier to maintain; it can be viewed as a small component of the total system (essentially the core of the OS).

![Figure 3.1: Monolithic kernel vs microkernel.](image)

The microkernel must allow the development of OS services on top of it. It should provide only the basic features, such as some way for dealing with address spaces, for manipulating memory protection; some entity that represent the execution in CPU, usually the tasks or threads; and a inter-process communication (IPC), needed to invoke the servers. This minimal design was introduced in [62], although the first time that microkernel-like concepts appear seems to be in the RC4000 Multiprogramming System in the 1960s [63] and the term microkernel appears in 1988, for the Mach kernel [64].
Nevertheless, for efficiency purposes, some microkernels include the scheduler in kernelspace. Many projects have attempted the kernel formalization, having in mind the certification. This degree of assurance goes in the path of EAL-7. With this kind of practices, the microkernel designs have also been used in systems made for high-security applications, like military systems.

However, these types of systems have various types of applications, some more important than others. For example, in a plane it is not hard to imagine that the application which controls the temperature is slightly less important than the applications that controls the engines. The solution was to have several computers, each of which controlling only a single application. In this way, if any of them fails or has unexpected behavior, there is not harm or change in the functioning of any other. However, if for each function we have different hardware, it can easily be seen that for certain systems this solution is inadequate.

Thus, the need for new solutions arises. This case is essentially a rediscover of the concept of separation kernel. At the time it was introduced, it was not given a great importance, because at the time there was no need for such architecture. However, times changed and the concept was reviewed.

### 3.2 Separation kernel

The need to run separate applications on the same hardware has increased the relevance of the concept of separation kernel. It is preferable to run several applications on the same hardware than having several hardwares, one for each application. There are great advantages in this approach. In some systems it is even impossible due to lack of physical space (for example in robots of exploration). The major disadvantage of having each application in an independent machine is the redundancy of resources. This leads to disadvantages such as higher costs, more space occupied and power consumed. An increasing of cooling, weight and more difficult installation and maintenance.

The concept of separation kernel was introduced by John Rushby in 1981 at the ACM Symposium on Operating System Principles. He introduced a new paradigm for secure computing by redefining the mission of the OS security kernel in same way as a distributed system. The new kernel simulates a distributed environment with only one processor and ensures that there is no interaction between the properties of the kernel.
Chapter 3. Case study: a secure partitioning microkernel

and the properties of the components that form such system. The separation between components enables independent verification of the kernel and its components.

The use of this type of kernel is based on the concept of separation or partition. The main purpose is to isolate faults of one component from the others. A failure or unexpected behavior in one partition (component) must not cause failure or unexpected behavior in another partitions.

The partitioning is divided in spatial partitioning and temporal partitioning. The spatial partitioning must ensure that software in one partition cannot modify the software of another partition. The address space of each partition is therefore isolated. Temporal partitioning must ensure that for all the partitions - one at a time - a time slot is given for shared resources, such as the CPU.

Until this point we used the term application to describe the computational entity within a partition. This term depends on the implementation. An application could correspond to the OS notions of process, or virtual machine, or other different notions. Generally, an application is composed by smaller units of computation that are called or scheduled separately. Again, this depends on the implementation, but generally these units are tasks or threads. Partitioning must prevent applications to interfere with each other, but the tasks within a single application are not protected from one another.

### 3.2.1 Spatial partitioning

The goal of spatial partitioning is to provide a mechanism to divide the memory, both physical and virtual. This enables assigning a block of memory to each partition. The addresses in this block are visible only for the partition owner of the block; all the other partitions cannot access this piece of memory. This ensures that the other partitions do not interfere in this memory space.

The most common mechanism used to guaranty that there are no violations of spatial partitioning is provided by the hardware, either by the processor running mode, the memory management unit (MMU), or a combination of both. The basic idea is that the processor has two modes of operation (it can possibly have more): when in user mode - the lowest privileged - the access to memory addresses are either checked or translated using tables in the MMU. These tables can not be modified. Only the kernel running in privilege mode can do it. The locations of blocks of memory in each partition
are disjoint as expected. The only exception are the locations used for inter-partition communications. The communications are discussed below.

The tasks executed in a partition access both the processor registers and the memory. This information is called the context of a partition. When one partition is suspended and another one starts, the kernel saves firstly the context of the partition being suspended and secondly reloads the context saved for the partition that is meant to be executed next.

### 3.2.2 Temporal partitioning

The temporal partitioning must guarantee that the execution of one partition does not disturb the timing of events in other partitions. A mechanism is necessary to ensure that a partition is executed in a dedicated time slot and that this slot of time will always be available for this partition, unaffected by the execution of the remaining partitions.

In our microkernel, the sequence of execution time given to each partition is statically scheduled to ensure determinism and simplification of the model.

The main problem to be solved is that one partition may get or keep exclusively possession of the CPU for itself. This can occur by bad intentions or simply by an error where an application is retained in an infinite loop.

The most straightforward manner to overcome this problem is by scheduling at two levels, as we can see in Figure 3.2: first, the kernel schedules the partitions; then, the partition schedules locally its own tasks. Usually, static scheduling is used at partition level. The kernel ensures that the partition runs for a determined time at a specific frequency (for example runs 10 ms, every 100 ms). Then, the scheduling inside a partition is dynamic - based on common priorities of tasks. This gives the partition that is running the illusion of exclusive access to computing resources. The concept of paravirtualization has the same principles, where each partition is a virtualized OS, such as Linux or Windows.

### 3.3 Security partitioning and MILS

Besides space and time partitioning, and due to the need of information exchange between partitions, the necessity of information flow control emerges.
The MILS (Multiple Independent Levels of Security/Safety) architecture was introduced in 1983 [66]. The authors, John Rushby and Brian Randell, recently discussed how this architecture appeared [67]. This architecture adopts the best principles of security and safety-critical design. It is based on the concept of a small separation kernel and was proposed to be evaluated to the highest levels of security, namely EAL-7 and safety assurance DO-178B.

The MILS provides the following properties:

- **Information Flow policy**: only authorized communication between partitions is allowed (pre-configured communication channels);
- **Data Isolation policy**: information in the state of one partition must not be accessible to other partitions;
- **Residual Information Protection policy**: a context switch between partitions with shared resources between them, can not allow unintended transfer of information;
- **Damage limitation policy**: failure in one partition is contained and does not disturb other partitions.

---

1Developed by the Radio Technical Commission for Aeronautics (RTCA) DO-178B [68], is a set of guidelines for the production of software for airborne systems. Designed to ensure that software meets airworthiness requirements, is a method of component approval in many critical aerospace, defense and other environments, including military, nuclear, medical, and communications applications.
As we can see in Figure 3.3, the MILS architecture is decomposed in three independent layers, a partitioning kernel, a MILS middleware layer, and a MILS application layer. As we have already seen, the partitioning kernel enables a well defined separation between partitions and a secure transfer of control between them, as we shall see below. The middleware allows application component creation by providing traditional OS middleware, such as CORBA, Web Services, and a Partitioning Communication System (PCS) for communications middleware for MILS. The application layer implements application-specific security functions, such as firewalls or cryptomod. The different levels of classification for distinct partitions are based on military classification of security levels classifications used by the U.S. Government, top secret (TS), secret (S), classified (C) and unclassified (U).

One of the first users of this technology was The Boeing Company [69], which has employed the RTI [70] and Wind River MILS solution [71].

The scheduling of partitions is static: this implies that all inter-partition communication must be asynchronous. Where a task needs the services of software in another partition, it places requests in the input buffer of task in other partitions and continues the execution. When the following partition is activated, it looks within its own buffers for replies or requests from other partitions. The process can be seen in Figure 3.4. It is necessary to impose fixed quotas on the number or space available for requests that can be made by each partition. This enables a great control and prevents the generation of excessive number of requests to another partition by malicious software.
Chapter 3. Case study: a secure partitioning microkernel

There exists a Common Criteria Protection Profile for separation kernels entitled “U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness”, commonly known as SKPP [72].

In SKPP the monitored information flow between partitions is made with the help of a Partition Information Flow Policy (PIFP). This is the same as channel in MILS architecture. The information flow is identified by a triplet, which consists of two identifiers of partitions and the availability of communication. The PIFP is based on the principle of least privilege (PoLP): this constitutes a crucial element in the design of high assurance systems. The PoLP minimizes the accesses between the entities: each component of the system must have access only to the resources and data that it needs to perform its function or purpose. There are two levels of policies of communication. With a higher level of abstraction, it is assumed that all tasks in a partition have the same needs to access resources in the exterior. This restricts the management of communication at partition level, since all tasks in a partition can only do the same kind of communication. Alternatively, the communication can be made in a more detailed way. The exchange of information can be done between tasks of partitions. In this way, tasks within the same partition have possibly distinct privileges for communication, as we can see in Figure 3.5.

The criticality of the application and the requirements of their security may require high assurance levels. An example of this is an avionics system, such as a GPS receiver for military purposes that requires safety and security approvals. In the past, these systems were implemented using software on a dedicated hardware, ensuring that information could not be shared outside the system. Nowadays, with architectures such as separation
kernel and MILS, both classified and unclassified functions can be executed in a single processor, with the desired level of security.

One of the leading OS in the military and aerospace markets, where reliability is absolutely critical, is Green Hills [73]. They provide a set of products according to the demand of customers. For example, the Integrity-178B OS is planned for the F-35’s core processor and is slated for an upgrade to the F-22’s integrated core processor. The Integrity-178B is designed to meet the EAL-6+ standard [74].

Other important company that provide good solutions for the safety-critical real-time systems is LinuxWorks [75]. Its LynxSecure 3.0 separation kernel and hypervisor has been designed to be certifiable to the Common Criteria EAL-7, and complies with the aerospace industry’s DO-178B certification.

Another successful case, is PikeOS developed by SYSGO [76]. It is a real time OS supporting the principle of software partitioning, widely used in defense, aerospace, automotive, and industrial applications. This was completely developed according to the development process requirements of the DO-178B and IEC 61508 specifications.

---

3.4 Summary

As seen throughout this chapter, the main characteristic of a microkernel is to have some OS services running in user mode. This makes the microkernel simpler, safer, and smaller, when compared with the normal kernels (monolithics). The microkernel is therefore the ideal candidate to serve as the basis for the separation kernel, which arises from the need of a system having to execute - in the same hardware - different tasks independently and securely. We also introduce the secure partitioning microkernel that, apart from containing space partitioning and time partitioning - which are included in most recent microkernels - also contains secure communications, thus making it different from others.

As can be seen in this chapter, building a secure partitioning microkernel is not an easy task. The solution proposed to build the system - as mentioned previously - is based on the CbyC development method. The work consists of the following steps:

1. To survey the requirements for the analysis of the system - \textit{i.e.} obtaining the necessary information to move forward towards the system implementation goal;
2. To build a high-level specification in a formal language (Z notation) - this step is important for a full understanding of the features and properties of the system;
3. To use the INFORMED process in order to obtain a cleaner and more consistent programming, removing repeated code and unnecessary exchange of information;
4. To implement the system in SPARK;
5. To verify the system (using the SPARK Examiner toolset).

Before undertaking this, we will review in the following chapter similar applications implementing formalization and verification of kernels that are available from the literature.
Chapter 4

Formal approaches to kernel development

There are many works in the literature on the subject of formalizing Kernels. This is understandable because the kernel is the central core of any computing system and its proper functioning is essential. The use of FM is of utmost importance because it allows us to have a formal specification of a kernel that represents an unambiguous description of its features (requirements) for the developers. A formal specification of a kernel also serves to provide the basis for the verification of an implementation of the kernel. The most usual technique is to prove that a formal representation of the implementation is equivalent to the top level specification. Normally, the use of simple formal notations and various levels of refinements aim to facilitate either manual proof or the use of an automatic theorem prover. On the other hand, there is also the possibility to verify directly on the implementation level without a more abstract specification of its behavior. The works in literature related to the verification of kernels present various levels of abstraction and various levels of completion. More recent projects, which achieved satisfactory results, have benefited of the constant evolution of tools. For example, the automation achieved with current theorem provers is a fundamental help in the verification process, avoiding (at least in part) a tedious and very specialized job.

In this chapter, we present the related work to formalization of both traditional and separation kernels. Besides this, a more interested reader may consult the article, which
served as the main reference for this chapter, on past and present approaches to kernel verification [77]. This chapter includes some works described in that article, though in less detail; it also describes other works that seem relevant in the context of our work. In the penultimate section we present the Tokeneer project that was of great relevance in the choices made in our work.

4.1 Traditional kernels

Earlier work on OS kernel formalization and verification includes Provably Secure Operating System (PSOS) [78] and UCLA Secure Unix [79]. The focus of these works was on capability-based security kernels, allowing security policies such as multi-level security to be enforced. These efforts were hampered by the lack of mechanization and appropriate tools available at the time, and so, while the designs were formalized, the full verification proofs were not practical. Later, with the emergence of the first support tools and automations of processes, all the works can achieve a new level of verification. For example, KIT [80], which appeared some years later, describes verification of properties such as process isolation to source or object level. Although it is a simple kernel, it is nonetheless significant the amount of properties that could be proved during the project.

The PSOS [78] project began in 1973 and continued until 1983. PSOS not only focuses on the kernel but on the whole system, including new hardware design. The system contained new hardware and it is unclear which percentage of the system was implemented, however the design seems to be very complete. The authors said that no code proofs were done, only a few simple illustrative proofs were carried out. However, PSOS is a project with an impressive effort that pioneered a number of important concepts for OS verification and system design in general. Formal methods were applied during the whole implementation of the system. PSOS has a layered architecture whose design comprises 17 layers. The bottom six were intended to be implemented by hardware. PSOS is not a kernel-based system, instead it is based on the principles of layer abstraction, modules, encapsulation, and information hiding. PSOS uses a capability mechanism that provide a controllable basis for implementing the OS and its applications, as there is no other way of accessing an object other than by presenting an appropriate capability designating that object. For designing PSOS, the project initially developed the Hierarchical Development Method (HDM) [81] with its specification and assertion language SPECIAL. Each system layer is composed by some number of encapsulated modules with
a formal interface and each module was formally specified in SPECIAL. Some specifications evolved up into the application level, including a confined-subsystem manager, secure e-mail, and a simple relational database manager. In a retrospective report [82], the authors claimed that the PSOS architecture effectively eliminated the popular myth that hierarchical structures must be inherently inefficient. The design methodology of PSOS was later used for some projects like in the Kernelized Secure Operating System (KSOS) [83, 84] and in The Secure Ada Target (SAT) [85].

More recently, Verisoft project [86] also made an effort to verify the whole system, including hardware, compiler, applications, and a microkernel. The project started in 2003 and was funded by the German government. The verification is made in a layered approach similar to PSOS. One of the layers establishes a hardware independent interface [87] which is very convenient for verification purposes because it isolates the parts of the kernel that involve Assembly code. The system, through a simple OS, provides file based input/output, IPC, sockets, and remote procedure calls to the application level. The goal of the project was to end with only one final machine-checked theorem on the whole system, including devices. This theorem has not yet been published and the current proof state covers about 50% of the code - however, recent publications appeared to go in this direction [88, 89]. Even without future results, this project constitutes an evidence that with current technology state it is possible to obtain a trustworthy basis for computing systems. Nevertheless, there are some issues that can not be ignored. One of them is the fact that the project focuses only on implementation correctness and did not investigate high-level security policies or the access control models of the OS; another one, is the fact that the system is only available for the VAMP processor, and the performance is not at the level of acceptable, because, one more time, it was not a focus of the project.

The Verisoft project besides using a layered architecture as in PSOS, where each level is implemented by different code, also uses a layered architecture at each layer. That is, each layer has one or more specifications (formalizations) that differ only in the detail level. This technique is called data refinement and consists of map existing functions between the layers in order that an operation in the most abstract layer has the same effect as that in the more concrete layer. The first project using this technique (although at that time still did not have this name) was the UCLA Data Secure Unix [79]. It is an OS that was aimed at providing a standard Unix interface to applications. Its verification was focused on the kernel of the OS, which by its features is close to the services of modern microkernels. UCLA provides threads, capabilities (access control),
pages (virtual memory), and devices (input/output). The report does not specify when the project started, but the first results began to appear in 1977. The specification of the project was divided into four layers. Starting from the top, the “top-level specifications”, the “abstract-level specifications”, the “low-level specifications”, and the “pascal code” at the bottom. The authors concluded that code proofs, with the aid tools available at the time, represented tedious hard work. They proved less than 20% of the code, with XIVUS semi-automated verification system, and they recommended the separation of the system development from the proofs. Nevertheless, they said that the system needed to be developed with verification in mind. They also concluded that the time spent in specification, verification, and implementation can be less than the time spent on design, implementation, and debugging. This is because the time spent on testing and validation corresponds to approximately 70% of the total time spent on development of the system.

Almost a decade after PSOS and UCLA Secure Unix, KIT appeared [80, 90]. This is a small OS kernel written for a uni-processor computer with a simple Von Neumann architecture. It provides isolated tasks as its main service (like its name “Kernel for Isolated Tasks” suggests), and also provides access to asynchronous I/O devices, exception handling, and single-word message passing. Concerning memory, KIT does not provide shared memory or virtual memory. The system is implemented in an artificial but realistic assembler instruction set. Even with its simplicity, KIT is important because it was the first kernel that deserved the attribute formally verified, breaking down the idea that the level of detail required in OS implementation verification is an intrinsic problem for formal verification. The verification was carried out in the Boyer-Moore theorem prover [91], the predecessor of the ACL2 prover [92]. Very similar to UCLA Secure Unix and other refinement-based verifications, the proof of the KIT system shows correspondence between the behavior of finite state machines. The corresponding proof shows that the kernel correctly implements this abstraction in a single CPU.

Later on [93], Bevier and Smith also produced a formalization of the Mach microkernel [64]. They specified legal Mach states and described Mach system calls using temporal logic, but they did not proceed to implementation proofs. Nevertheless, this work served as basis of a new approach to specification-based testing [94]. Here the authors used the new approach to check properties of MK++ kernel [95], a descendant of the Mach kernel. They suggested that a mathematical proof of an implementation model satisfying the specification is impractical given the complexity of the kernel implementation. Therefore, they chose the approach of exploring the derivation of tests from the kernel
specification, with the certainty that, if a test fails, the specification is incorrect. This technique is similar to model checking, that generates a sequence of steps, which proves the incorrectness of the specification, if some property is violated. However, normally, the absence of this counter-example does not allow to ensure that the specification is necessarily correct. As seen previously, in Section 2.1, this lack of assurance is due to the inability to check the whole system. The complexity of the system leads to an explosion of states which make it impractical because the computation time needed (if possible) is unacceptable. So, the model checking is commonly applied only to specific parts of a system.

In RUBIS kernel, G. Duval and J. Julliand used the technique above to model and verify the entire inter-task communication features of the kernel [96]. The approach taken consisted on communicating finite state machines. They used PROMELA as specification language and the SPIN tool [38] to check the intertask communication features of the system. They constructed various scenarios to test the properties of communication, with the “confidence” that if the scenario did not produce any error the communication mechanism test did not present errors.

The advantages of model checking are again exploited by a concurrent system with similar characteristics to a microkernel [97]. For the verification of the system, the project used a combination of TLA+/TLC as specification language and as model checker respectively. The principal focus of this work was the resource management mechanism and the protocols to ensure the consistency of the data in those shared resources. Resources are mutually exclusive to ensure the consistency of data in shared resources. In order to accomplish the objective of a real-time kernel, a priority handling mechanism was presented. At the end of the work, a set of solutions for real time OS were proposed.

Another project that focused on only a single aspect of the kernel, was SELinux (Security-Enhanced Linux) [98]. The project provided a mechanism for supporting access control security policies as a Linux feature. It was based on the Flask architecture [99]. Flask is an OS security architecture that provides flexible support for security policies. The architecture was prototyped in the Fluke [100] research OS. Later on, it was taken by the NSA to Linux as the security architecture in SELinux to transfer the technology to a larger developer and user community. It was made subjected to two different approaches: one used TAME (Timed Automata Modelling Environment) [101] which is based on the PVS prover [102]; the other, used model checking [103]. The two projects analyzed the security policies themselves, but did not aim at proofs establishing that the SELinux kernel correctly implements them.
The kernel verification of the following three projects has the data refinement approach in common. In the first one [104, 105], the authors offered an abstract model of the functional and timing requirements for the kernel. It was designed to provide the minimal functionality to support real-time Ada 95 applications closest to Ravenscar Profile. The kernel was specified using the PVS specification language, with the temporal properties expressed using a stylized version of RTL\(^1\) [106]. The kernel was specified in terms of its state and a set of operations on that state. The structure of the specification provides a clear distinction between the kernel and its environment, and defines how they interact, nonetheless they not introduce memory. The sample implementation - in Ada - provides the main features of the kernel as follows: the fixed priority preemptive scheduling of tasks, delay operations, and asynchronous communication between tasks. For the high-level operations, they used SPARK annotations. They did not use the full SPARK annotations but they were verifying the functional correctness of the operations with respect to the previous level of the development. The development method that was used here, as we mentioned before, is based on splitting the development process into a number of stages, and verifying the correctness at each stage, in a similar fashion to VDM [28] or to B-Method [21]. At each stage, both the temporal properties and the functional properties of the system are verified in relation to the previous stage in the development. This enables a gradually abstraction decrease in the specification until the final implementation is produced.

The second, is the delta-core OS [107]. It appeared as a refinement of an initial formal specification. The formal specification was proved with the help of PowerEpsilon [108] (a mathematical proof development system). The proofs achieved represent some important characteristics of the OS. They verified some system calls for tasks, queues and semaphores.

The last of the three is a very helpful work when using FM to design, verify and implement kernels. It used Z notation to present the concepts related to the kernel functionalities in a fashion and comprehensive way [109]. The formal models of three OS kernels were presented and some important concepts that sometimes are not addressed, like hardware abstraction model, virtual storage, and interrupt service routines (ISR), were also presented. The first model was a simple kernel, such as those that are used in embedded and real-time systems. It was a basic kernel and does not dealt with ISR or device drivers. The second kernel was an extension of the first one. It adds the device drivers and a clock for the process-swap mechanism. Inter process communication (IPC)
was implemented using shared memory and semaphores for synchronization control. The last kernel modeled was a variation of the previous one. The difference was that IPC was now implemented as message passing, using ISR. This required changes to the system processes, as well as the addition of generic structures for handling interrupts and the context switch. The kernels properties were provided and the proofs of the correct behavior were included. At the end of the book, a model for virtual storage was presented. The initialization and refinement of models were not covered in this book. However, the author published a second book which deals with refinement of models. The new book is presented in the next section because it includes a separation kernel.

Instead of designing and implementing new kernels, other works undertook the verification of existing ones. The two following projects, EROS (Extremely Reliable Operating System) and L4, show this. For EROS verification [110–112], the authors gave an operational semantics of the OS and proved a correctness of its architecture with respect to confinement security policy. They developed a formal statement of requirements and a simplified model, Agape (more powerful and more general than the EROS architecture). They modeled Agape’s security policy, access control mechanism, and operational semantics, and shown that Agape semantics satisfies the requirements of confinement. Their proof provides a small number of essential lemmas that must be satisfied for any system to provide confinement, and should generalize to other capability systems. The EROS capability system defines an access control mechanism that determines what information flow is possible between system resources. The model was not formally connected to the EROS kernel implementation. This was supposed to change for its successor the Coyotos kernel [113].

In Coyotos project the intention was to carry out verification since the very beginning. They changed the approach and the project laid out a plan for the design and formal implementation of the new kernel. For this, they identified the need to create a new programming language for kernel implementation - one language that was safe, clean, and suitable for verification and low-level implementation at the same time. The main goal was not necessarily to invent new language features, but rather to pick and choose from existing research and combine the necessary features into a targeted language for implementing verified OS kernels. The reasons for this effort lies in: a) the difficult of the verification of programs written in the main languages, like Assembler, C, and C++ with its many unsafe features; b) the benefits in future verification efforts. The Coyotos project made significant progress on the design and implementation of the kernel itself.
Chapter 4. Formal approaches to kernel development

until 2009. The design of BitC, the proposed new implementation language, has taken
advances more recently.

For the verification of L4 \cite{62}, two different projects appeared: seL4 (secure embed-
ded L4) and L4.verified \cite{114, 115}. The first project was a descendant of L4 and
aimed to provide security improvements to communication control between applications,
and to kernel physical memory management. The second project aimed to provide a
machine-checked formal correctness proof of a high-performance implementation of the
first one. After a small pilot project in 2004, seL4 and L4.verified started concurrently
in 2005 \cite{116}. The initial pilot project resulted in initial design ideas for seL4 \cite{117}, in
a case study on virtual memory of the existing L4 kernel (using the theorem prover Is-
abelle/HOL) \cite{118, 119}, and in a high-level specification of L4 IPC (using the B Method)
\cite{120}. The seL4 project was concluded successfully by the end of 2007 and the result-
ing design provides the following kernel services: threads, IPC, virtual memory control,
capabilities, and interrupt control. The capability system of seL4 is similar to that of
EROS kernel and are not dependent on hardware, like in PSOS. The success of seL4
kernel was due in large part to the good integration and cooperation of both OS and
FM teams. They found a good balance, not privileging the OS team that would tend
to improve only the performance of the kernel, leaving the verification more difficult, or
the opposite, making verification easier but neglecting the details of performance, like in
Verisoft project. With this cooperation, they maintained the size manageable, contrary
to what happened in the PSOS. The L4.verified project was able to prove that the seL4
microkernel works correctly. Yet it was not able to prove that the seL4 is secure enough.
For instance, it does not imply that two isolated subsystems, each of which does not
possess any capabilities to the other nor to any shared resources, cannot send each other
information, either directly or indirectly.

A different approach from what we saw until this point, was carried out on VFiasco
(Verified Fiasco) project \cite{121}. The project started in November 2001 and its aim was
verifying parts of the Fiasco microkernel (a compatible re-implementation of the L4)
directly on the implementation level (source code), without a more abstract specification
of its behavior. The implementation language was C++ with isolated interface code and
optimized IPC in Assembly. The approach followed in this work was a precise formal
semantics of the implementation language. C++ is a very large and not a very clean
language, it was not designed with verification in mind. The authors proposed a clean
semantics of C++ (Safe C++) that deals with those undesired features of C++ that
were used in the Fiasco sources. Besides it used model checking (SPIN) for safety
properties on the IPC, the verification was made in the interactive theorem provers Isabelle/HOL and PVS. They proved some object-store properties, such as: a) writing to some allocated object does not accidentally modify any other allocated object; b) after writing to an allocated object, reading from that object actually returns the value written; and c) the order in which you allocate or deallocate objects is irrelevant as long as you deallocate objects with the same allocator used in the first place. Nevertheless, the verification was achieved only by a small portion of the implementation.

The ROBIN (Open Robust Infrastructures) project continued the VFiasco approach to C++ source code verification [122, 123]. The project started in February 2006 and aimed to investigate the verification of the Nova hypervisor, an L4-based kernel, this is a different microkernel for OS virtualization. The project ended in April 2008 and produced about 70% of a high-level specification for the Nova kernel [123]. Like what happened with VFiasco, this project also not achieved a verification of a considerable part of the implementation.

This section is summarized in Table 4.1. The first column shows the name of the projects. The next two columns show, respectively, the year when the project was made (years between parentheses represent an estimation) and the specification language used in the project. The fourth column identifies the use of model checking technique, and the next two columns identify the amount of proofs achieved in the project and also the theorem prover used for that. Finally, the last column shows whether the project achieved the implementation level. The projects included in this table have different purposes: a) PSOS and Verisoft aimed to verify the whole system; b) the complete kernel verification was the purpose of UCLA Secure Unix, KIT, and L4.verified/seL4; c) the remaining projects aimed to verify only specific parts of the kernel. Another aspect, is the use of data refinement approach; this method was followed by UCLA Secure Unix, KIT, Verisoft, and L4.verified/seL4; this method was not used in the remaining projects.
Table 4.1: Traditional kernels verification - adapted from [77].

### 4.2 Separation kernels

As seen in Section 3.2, the separation kernel was introduced in the early 80’s [65, 124], but only a few years later - given the need of such system - the concept was rediscovered. Since this is a more specific type of kernel, there is less work available in the literature. Below we will present some of the most relevant work reviewed.

One of the first works that appeared in the literature was a safety kernel for traffic light control [125]. The work used a realistic example to show the possibility of implementing a Rushby kernel. Properties of the system were specified in Z notation and an implementation of the kernel written in Ada was proposed.

The Mathematically Analyzed Separation Kernel (MASK) [126, 127] emerged in the current century. The MASK was designed and built using Specware [128]. The authors started from high-level model and down to reach a low-level design, which was close to an implementation with multiple formal refinement proofs. This low-level design was manually translated into C and reviewed against the Specware models. Its main
application was a cryptographic smartcard platform developed at Motorola. The project was a collaboration between the NSA, Motorola, and the Kestrel Institute.

The delta-core OS, presented in the previous section, was extended by the same authors with a partitioning optional feature [129]. The authors presented a formal specification of partitioning and also presented the mathematical properties to provide assurance for partitioning. The resulting specification contain the original dynamic scheduler for tasks and increment a static scheduler for partitions. Like in the first specification, PowerEpsilon was the language chosen for formalization. The specifications for the address space were ignored.

The next work was a little different from all those presented until this point. It concerned a verification of a microprocessor with intrinsic partitioning mechanism [130]. This was not a kernel verification because a kernel is by definition a software, but the policies and properties considered are very similar and closely related. The project concerned the verification and Common Criteria EAL-7 certification of the AAMP7 microprocessor. This is a microprocessor designed for use in embedded systems with security-critical or safety-critical requirements. The AAMP7 provides a novel architectural feature, intrinsic partitioning, that enables the microprocessor to enforce an explicit communication policy between applications. The implementation language is processor microcode and they used ACL2 to show that the AAMP7 microprocessor works as expected. They modeled the implementation of the AAMP7 with respect to partitioning in great detail and the model corresponds directly to the microcode of the microprocessor. The proof was based on a generic high-level model of separation kernels proposed in an early work [131]. In this, the code was translated into ACL2 automatically, opposed to the AAMP7 verification where the code was translated manually.

Still, having EAL-7 level as its objective, another work emerged. It proposed a novel approach for verification and Common Criteria certification of a software-based embedded device, featuring a separation kernel [132] [133]. The authors did not specify which device, which kernel, and which evaluation level exactly, but the report mentions 3,000 lines of C and Assembly code as the size of the kernel and the proofs presented should qualify EAL-7. They used TAME for verification and they proposed some steps to assure that the kernel enforces data separation. To start with, a Top Level Specification (TLS) of the kernel was done using the style introduced by a previous work [134]; until they reached the phase of mapping this TLS to the source code. This project was a clear demonstration that formal methods can be used with a high cost/benefit for verification of these kind of systems.
In [135], the author presents a continuation of what was already presented in the previous section [109]. In this new book, the author presents the specification, design, and refinement for the executable code of two operating system kernels. Proofs for the refinement are presented to show that it is possible to refine and achieve code through refinement. An advantage in presenting the proofs is that it is possible to conclude that if the specifications are correct then consequently the refinements are correct too. Two models of kernels are presented. The first model is, like in the previous book, a small kernel that can be used in embedded systems. The other model is a separation kernel as proposed by John Rushby, approached in the beginning of Section 3.2. This is an important reference in our work because Z specification of a separation kernel is in fact one of the objectives of this dissertation.

Producing an embedded microkernel using the B Method was proposed in [136]. This last work was partially developed simultaneously with our work. Experiments were shared between both works and some requirements of the case studies are the same, because they were elaborated together. The author divided the work in three stages. First, it provided a complete requirement analysis and a specification of a secure partitioning microkernel. For this stage, he used the Atelier B and ProB tools. The second stage was composed by a complete development of part of the secure partitioning microkernel, starting with a high level specification, through successive refinements until the automatic generation of code was possible. In this stage, the partitioning information flow policy (PIFP) was the choice. The last stage of the work was the integration of the code generated in the previous stage with a chosen microkernel. The microkernel chosen was Prex (Real-Time Operating System). Although the code added to Prex was not formally proved, the test results were encouraging. The author concluded that it was possible to achieve the complete development of the secure partitioning microkernel using only the B Method. However, it is possibly easier if different FM are used, according to the specific part to be developed.

Only commercial projects of verified separation kernels could reach the implementation level. Nevertheless, these cases are not included in this review given the lack of information referring to such systems.
4.3 Tokeneer project

As mentioned above, the methodology used in this MSc work was based on the methodology that was used in the Tokeneer project, in a more simple way though imposed by our somewhat limited experience to perform the various steps. The Tokeneer project was carried out by Praxis and consisted of a re-development of the original work developed by the NSA (National Security Agency).

The project arose from a proposal made by the NSA to Praxis with the following objectives: a) to demonstrate that it is possible to develop highly secure systems in accordance to the highest security levels of Common Criteria; b) to show that it is possible to develop safe systems of a rigorous manner and simultaneously in a cost-effective manner.

The system consists of a secure enclave with controlled physical entry. Within the enclave, there are various workstations and the users must pass security tests in order to access the machines. The security tests are done with the presentation of a token (e.g. smart-card) into a reader that is outside the enclave. The system uses the information of the token to carry out biometric tests (e.g. fingerprint reading) of the user. If both informations are matched, then the enclave door opens giving access to the user. At the same time, the system checks what type of access is allowed for this user. The physical devices are replaced by test drivers to avoid licensing issues and to ease testing. They were developed by SPRE Inc. on a separate machine.

The development of the system was made according to Praxis CbyC process, containing the following phases:

1. Requirements analysis (the REVEAL process);
2. Formal specification (Z notation);
3. Design (refinement of the specification and INFORMED process);
4. Implementation in SPARK;
5. Verification (SPARK Examiner toolset);
6. Top-down system testing.

The Tokeneer project material was released in July 2008 as a contribution to the Verified Software Grand Challenge. It achieved EAL-5 in areas of development like configuration
control, fault management, and testing. In other development areas, as specification, design, implementation, or correspondence between representations, it achieved up to EAL-6 or EAL-7.

4.4 Summary

Substantial work has already been done in the verification of kernels. The verification purposes change from project to project: the verification of the whole system, the complete kernel verification, or verification of specific parts of the kernel. In some instances the kernel already existed, in others the approach was developed from scratch. As to the data refinement approach, this also adopted by only some works. With respect to verification of separation kernels only commercial projects could reach the implementation level given its specificity in both their properties and their utilization.

From this chapter we can draw some useful conclusions regarding the development of verification of kernels, and ultimately of system verification in general:

- the system needs to be developed with verification in mind;
- the system performance should not be overlooked in favor of the verification;
- if the system has a strong low-level (hardware) component, it is important to maintain a healthy balance between the requirements of the formal methods and the hardware for verification and implementation issues respectively.
- at last, with the automatism of nowadays tools, FM can be used with a high cost/benefit for verification of complex systems.

As stated in the aim and objectives of this work (see section 1.3), next chapter - on the implementation - shows how Tokeneer CbyC methodology was followed in our work, and how far the indications above were achieved.
Chapter 5

Implementation

In this chapter we show the implementation of the case study presented in Chapter 3 with the SPARK and the method proposed above in Chapter 2.

The project is based on an existing proposal and hence the functional requirements had been almost defined. Even so, some changes had to be made in order to meet our specific needs. Only the fundamental security requirements of the system are emphasized in here, with special relevance to the three main properties described in the previous chapter:

• Spatial partitioning: each partition can only access its own memory (information), the remainder memory is inaccessible;

• Temporal partitioning: a partition controls only the hardware during a given time. Once this time is over, the partition has no longer control over the hardware, which will be then controlled by another partition;

• Security partitioning: the communication between partitions is established safely. Communication’s contents are only shared by partitions involved in the communication process.

The list of the requirements is presented in Appendix A.

After the survey of requirements and with a set of requirements well-defined, we passed on to the implementation following the development cycle proposed above and shown in Figure 5.1.
5.1 Formal specification/design

In this section, we present the formalization of the requirements.

The core functions, based on the requirements, will be formally specified and modeled using the Z notation, a mathematical notation accompanied by an English narrative. The aim is to delineate unambiguously what the system is conceived for. After an initial version, this was developed in parallel to the INFORMED design stage.

The specification models a number of state components and a number of operations that change the state of the system. The entire model as been type-checked with the fuzz type checker [17] in all expressions and gave no errors.

The proofs on the specification were not carried out, but the behavior will be checked with a model-checker. To achieve this, we had to withdraw some abstraction from the models, using concrete representations for the system state. Some things are still non-deterministic, allowing a choice between two actions given the previous stage; others have concrete states in order to define the priority of actions so that they can be admitted and simulated with ProZ. This tool, as we present above in Chapter 2, is an extension of the B-Method tool[1]. The behavior of the system is well visible using animation and is a great support for the development. We obtained a model with intermediate

---

1 The ProB tool [19] is an animator and model-checker for the B-Method, however it also supports CSP-M and Z notation.
characteristics between abstraction and concrete, which can be subjected to evaluation of their behavior. Only the necessary aspects have been passed on to the concrete, everything else remained with a high level of abstraction and low implementation details to facilitate understanding.

As previously stated in the explanation of Z, the operations in most cases involve several phases, each of them representing a small action. For example, the Send operation involves a verification of the authorization to communicate (specified by the schema CheckPIFP) and only after the success of this operation is possible a real communication (specified by the schema ComSend). The overall process of sending can then be presented as the combination of these two schemas. The system description is generated through this way.

A summary of the specification is presented below. For the complete description of the system models, the reader is referred to Appendix B.

The system state is represented by the Kernel schema shown below.

```
Kernel

Status : ReturnCode
FirstADDRAvailable : N
FreeMemory : N
Memory : seq Block
Clock : N
Mode : WorkingMode
Partitions : P Partition
CurrPartition : N
PIFP : P Policy
Communications : P Communication
```

The kernel is represented by:

- Status: which represents the state of system and is updated by the operations;
- FirstADDRAvailable: which assumes the first address of free memory;
- FreeMemory: which assumes the size of free memory;
- Memory: which represents the physical memory and is constituted by a sequence of blocks;
- Clock: which is the representation of the elapsed time;
• Mode: which shows the working mode allowed at a given moment;

• Partitions: a set with the system partitions;

• CurrPartition: which represents the running partition;

• PIFP: a set that contains the communication policies for all system partitions;

• Communications: a set that contains the communications performed by the system.

The enabled operations are:

• Start: this performs an initialization of the system;

• ContextSwitch: which changes the execution partition of the system when its time is exhausted;

• ReadWrite: which performs a read or write operation (it tries the access to a memory space);

• Send: which performs a communication to send a message;

• Receive: which performs a communication to receive a message;

• Tick: as the purpose to give a tick of the clock CPU generating an interruption and, changing the kernel to privileged mode.

The main differences between the specification hereby proposed and some of the ones available in the literature (e.g. \cite{109, 135}), is the simple way in which they are developed and yet still covering the requirements for the purposes established. The communication mechanism implemented in our system has its own particularities different from the others. These have to do principally with the need to meet a pre-established communication security police.

Ideally, we should have two types of models. First, a formal specification of the system with black-box behavior; second, a formal design which is a refinement of the first model, but with implementation details. The objective is to separate the external visible behavior of its internal design. This was not the approach followed since we wanted something simpler and more high level for the perception of the system and its behavior.
With the completion of this step, we could improve knowledge of the system and get some of its peculiarities. This was a step of great value towards the final result because it allowed us to obtain a general view of the system, with the elimination of ambiguities in its features.

5.2 INFORMED in practice

Here, we present aspects of the design process that are not covered by the Formal Specification/Design, but that are required in order to make progress for the implementation in SPARK. The INFOMERD helps to shape the architecture in terms of Ada packages, defines types and operations provided by these packages and relates this with the formal model. Furthermore, this step covers the file formats and file locations used by the system.

5.2.1 Identification of the system boundary, inputs and outputs

The real world (or at least, the real peripherals) that are outside the kernel, will be emulated. When the system starts, the configuration file provides input to the kernel. It is up to the kernel to then respond by reading the real world input into its own internal representation. The kernel receives stimulus from the real world and it changes itself the real world. All real world entities are modeled as components of the real world.

The real world entities modeled are the memory and the configuration file, as shown in Table 5.1. Only the memory changes with the execution of the system. The configuration file represents the configuration chosen for the kernel.

Many components, like keyboard, screen, network card, etc, are dismissed because they increase the complexity of the entire system.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Input / Output</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>input / output</td>
<td>The kernel uses the memory in the configuration process. The memory also can be updated as part of the normal execution of the kernel.</td>
</tr>
<tr>
<td>configuration file</td>
<td>input</td>
<td>Data from the configuration file is used by kernel for configuration of the system at the initialization process.</td>
</tr>
</tbody>
</table>

Table 5.1: System boundary, inputs and outputs.
5.2.2 Identification of the SPARK boundary

Figure 5.2 depicts the mapping of variables of the system boundary in SPARK packages, delimitating the SPARK system boundary. The variables of the system boundary are as follows, as seen in the previous section: the memory (Hardware), which was mapped in the Memory SPARK boundary variable, and the configuration file (FileData), which was mapped in the File SPARK boundary variable.

5.2.3 Identification and localization of system states

At this stage, we need to identify the state that needs to be saved. This is one of the most important stages of the entire development, because these choices have a major influence in the amount of information flow in the system. The System is divided in various packages. The principals are SYT (System Table) that contains the tables with the state of the system, SEF (System Error and Faults) that contains the health condition of the system, Hardware that simulates the real hardware, ConfigValues that is important for the initialization of the system, CMS (Configuration Management System) that configures and maintains the system, and PRT (Partition) that facilitates the creation and manipulation of partitions. Table 5.2 describes the packages that keep the state.
Chapter 5. *Implementation*  

<table>
<thead>
<tr>
<th>State</th>
<th>Constituents</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYT</td>
<td>Partitions Table</td>
<td>Saves the information concerning partitions that belong to the system.</td>
</tr>
<tr>
<td></td>
<td>Communications Table</td>
<td>Saves the information concerning the communications, intended by the partitions.</td>
</tr>
<tr>
<td></td>
<td>PIFP Table</td>
<td>Saves the information concerning the PIFP policy.</td>
</tr>
<tr>
<td></td>
<td>PartitionsExecutionSequence Table</td>
<td>Saves the information concerning the partitions execution sequence.</td>
</tr>
<tr>
<td>SEF</td>
<td>Errors Table</td>
<td>Saves the information concerning the errors that occur in the system.</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>Represents the current state of the physical memory.</td>
</tr>
<tr>
<td>Hardware</td>
<td>Partitions State</td>
<td>A local copy of one partition which was read from the configuration file.</td>
</tr>
<tr>
<td></td>
<td>Size Of Blocks For Communication</td>
<td>The size of blocks for communication.</td>
</tr>
<tr>
<td>ConfigValues</td>
<td>PIFP Line</td>
<td>A local copy of one PIFP line which was read from the configuration file.</td>
</tr>
<tr>
<td></td>
<td>Partitions Execution Sequence</td>
<td>A local copy of partitions execution sequence which was read from the configuration file.</td>
</tr>
<tr>
<td></td>
<td>FileState</td>
<td>Saves the data from the configuration file at the initialization time.</td>
</tr>
</tbody>
</table>

*Table 5.2: State identification.*

The overall localization of state is given by Figure 5.3. As we can see, only SYT, SEF, Hardware, and ConfigValues are variable packages (packages with state). Then, we have the CMS and the File that are of the type utility layer and the rest of packages are abstract data types.
5.3 Handling initialization of state

State variables can be initialized using two approaches: a) during program elaboration (the variable is considered to have a valid value prior to execution of the main program); and, b) during execution of the main program by a program statement.

All state variables are initialized during program elaboration. Some of which are updated, specifically the memory state and the state variables from SYT which represent the concrete state of kernel. These will be updated by a procedure call of the main program. This procedure call has the responsibility to update the kernel state (tables) with information contained in the configuration file.

5.4 Handling secondary requirements

In this development, we not need to deal with any type of secondary requirements.
5.5 Implementing the internal behavior of components

The facilities provided by each of the library level packages in the system, the correspondence with Z models (that can be found in Appendix B), and the respective references to the requirements (that can be founded in Appendix A) are summarized in this section. We show the specifications of the packages. The specifications are presented with annotations, allowing an early static analysis of the design. The specifications presented contain the Ada signature for the operations and its annotations, which include the global state used or modified by the operation (the global clause) and the relationship between the global state and the operation parameters (the derives clause). Some functions and procedures are specified with proof annotations. After the specification packages are completed, body packages with implementation of operations are developed. Here, only the specifications are shown, the complete code can be found in Appendix C.

DefaultValues

In the DefaultValues package, there are some default values for the configuration of the system, such as the name and the path for the configuration file of the system or the value of memory size that can be handled by the system.

PartitionTypes

The PartitionTypes package contains the types needed for the partitions, which were used in the PRT package. All of the types are presented in Table 5.3 and are described below with more detail.
Chapter 5. Implementation

<table>
<thead>
<tr>
<th>Ada Type</th>
<th>Type Classification</th>
<th>Z Type</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition_Mode</td>
<td>enumeration</td>
<td>State</td>
<td>PRT#04</td>
</tr>
<tr>
<td>Communication_Mode</td>
<td>enumeration</td>
<td>CommunicationOP</td>
<td>CMS#03</td>
</tr>
<tr>
<td>Operation_Type</td>
<td>enumeration</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Processes_Type</td>
<td>record</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Partition</td>
<td>record</td>
<td>Partition</td>
<td>PRT#(01, 04, 09)</td>
</tr>
</tbody>
</table>

Table 5.3: PartitionTypes package.

- **Partition_Mode**: is composed by “IDLE”, “NORMAL” and “ERROR” values. Each partition has its state in one of these values;

- **Communication_Mode**: consists on the “FREE”, “READ”, “WRITE” and “BLOCKED” values. It allows to know the state in the communication space for a partition;

- **Operation_Type**: specifies the enabled operations, which can be “OP_Nothing” (when the purpose is literally to do anything, just to consume time), “OP_ReadWrite” (to read and write from/to memory), and “OP_Communication” (for both communications, send and receive);

- **Processes_Type**: is composed by an operation of the previous type and the values for these operation: in case of “OP_Communication”, it needs the size, the mode (send or receive), and the identifier of the recipient of communication; in the case of “OP_ReadWrite”, only the address and the size for the operation are needed. There is no need to know if it is read or write because there is no distinction between operations, we only need to know if there is permission to access that area of memory (i.e. if the space we want to access to belongs to the partition);

- **Partition**: is composed by an unique identifier, its memory bounds, the runtime allowed, its mode (which is one of the values of “Operation_Type”), a list of Processes (that will be run by the partition), and the state of memory spaces for communication (that will be the values of “Communication_Mode”).
PRT

The PRT package has the function to create and manipulate partitions. It has the operations summarized in Table 5.4.

<table>
<thead>
<tr>
<th>Ada Operation</th>
<th>Operation Type</th>
<th>Z Operation</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetID</td>
<td>function</td>
<td>-</td>
<td>PRT#04</td>
</tr>
<tr>
<td>GetDuration</td>
<td>function</td>
<td>GetDuration</td>
<td>PRT#03</td>
</tr>
<tr>
<td>GetPartitionMode</td>
<td>function</td>
<td>-</td>
<td>PRT#03</td>
</tr>
<tr>
<td>SetPartitionMode</td>
<td>procedure</td>
<td>PartitionResumes, PartitionSuspends, CurrPartitionError</td>
<td>PRT#03</td>
</tr>
<tr>
<td>Init</td>
<td>procedure</td>
<td>NewPartition</td>
<td>PRT#02</td>
</tr>
<tr>
<td>ClearCommunicationList</td>
<td>procedure</td>
<td>-</td>
<td>PRT#03</td>
</tr>
</tbody>
</table>

Table 5.4: PRT package.

Only the “partition mode” has a setter (Code Listing 5.2), this is the unique label that can be subsequently updated. The labels “ID”, “duration” and “mode” have a getter function (Code Listing 5.1) that retrieves its values.

```ada
function GetID(P : PartitionTypes.Partition) return PartitionTypes.IdType;
  --$ return P.ID;

function GetDuration(P : PartitionTypes.Partition) return PartitionTypes.Partition.Duration;
  --$ return P.Duration;

function GetPartitionMode(P : PartitionTypes.Partition) return PartitionTypes.Partition.Mode;
  --$ return P.Mode;
```

Code Listing 5.1: PRT specification package (GetID, GetDuration and GetPartitionMode functions)
Chapter 5. Implementation

procedure SetPartitionMode(P : in out PartitionTypes.Partition;
  Mode_Partition : in <-
PartitionTypes.Partition_Mode);
-# derives P from P, Mode_Partition;
-# post P.ID = P'.ID and
-# P.MemoryBounds = P'.MemoryBounds and
-# P.Duration = P'.Duration and
-# P.Mode = Mode_Partition and
-# P.Processes = P'.Processes and
-# P.Com = P'.Com;

Code Listing 5.2: PRT specification package (SetPartitionMode procedure)

As we can see in the Code Listing 5.3, the “Init” procedure is a constructor for objects
from “Partition” type. This procedure is used at the initialization of the system to
create the partitions specified by the configuration file.

procedure Init(P : out PartitionTypes.Partition;
  ID_Partition : in PartitionTypes.IdType;
  MemoryBounds_Partition : in PartitionTypes.Tuple;
  Duration_Partition : in PartitionTypes.Partition_Duration;
  Mode_Partition : in PartitionTypes.Partition_Mode;
  Proc : in PartitionTypes.Processes_List);
-# derives P from ID_Partition, MemoryBounds_Partition, <-
Duration_Partition, Mode_Partition, Proc;
-# post P.ID = ID_Partition and
-# P.MemoryBounds = MemoryBounds_Partition and
-# P.Duration = Duration_Partition and
-# P.Mode = Mode_Partition and
-# P.Processes = Proc and
-# P.Com = PartitionTypes.Communication '{PartitionTypes.Index_Range <-
=> PartitionTypes.BLOCKED};

Code Listing 5.3: PRT specification package (Init procedure)

The communication list has a procedure to clean its information. The “ClearCommunicationList” procedure, as we can see in the Code Listing 5.4, resets the state of memory
spaces for communication for a particular partition. This operation is executed always
before the partition receives the state of current partition.
procedure ClearCommunicationList (P : in out PartitionTypes.Partition);
  --# derives P from P;
  --# post P.ID = P'.ID and
  --# P.MemoryBounds = P'.MemoryBounds and
  --# P.Duration = P'.Duration and
  --# P.Node = P'.Node and
  --# P.Processes = P'.Processes and
  --# P.Com = PartitionTypes.Communication ' (PartitionTypes.Index_Range ←
        => PartitionTypes.BLOCKED);

CODE LISTING 5.4: PRT specification package (ClearCommunicationList procedure)

We can see in all operations above the SPARK annotations in blue. In our case, they are: the derives annotations for the procedures, and the proof annotations in functions with return clause or in procedures with post clause.

TablesTypes

The TablesTypes package contain the types used in SYT package. It has the operations summarized in Table 5.5.

<table>
<thead>
<tr>
<th>Ada Type</th>
<th>Type Classification</th>
<th>Z Type</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>PartitionsTable</td>
<td>array</td>
<td>-</td>
<td>PRT#(05, 07), CMS#(04, 05)</td>
</tr>
<tr>
<td>CommunicationState</td>
<td>record</td>
<td>Communication</td>
<td>-</td>
</tr>
<tr>
<td>CommunicationsTable</td>
<td>array</td>
<td>-</td>
<td>CMS#(03, 04, 05)</td>
</tr>
<tr>
<td>FlowMode</td>
<td>enumeration</td>
<td>FlowMode</td>
<td>-</td>
</tr>
<tr>
<td>PIFPTable</td>
<td>array</td>
<td>-</td>
<td>CMS#(02, 04, 05), PIFP#01</td>
</tr>
<tr>
<td>PartitionsExecSeqTable</td>
<td>array</td>
<td>-</td>
<td>PRT#(06, 07)</td>
</tr>
</tbody>
</table>

Table 5.5: TableTypes package.

The package contains the “CommunicationState” type that represents a communication made in the system. This type has the following labels:

- Blk: information of the memory intended for communication;
- State: current state of the communication (“FREE” or “OCCUPIED”);
• From_Partition and To_Partition: identifiers of the partitions involved in the communication;
• Mode: type of communication, i.e. send ("WRITE") or receive ("READ");
• FreeSize: free size of its memory (enables the use of all blocks of communication).

Code Listing 5.5: TablesTypes specification package
It also contains the “FlowMode” type, that allows the control of the communications; this is constituted by “R”, “W”, “RW”, “N”, respectively, read, write, read and write, and none.

The package contains four types of tables. The “PartitionsTable”, as we can see in Code Listing 5.5, is an array of the “Partition” type, its dimension depends on the number of partitions specified for the system. The “CommunicationsTable” is an array of the “CommunicationState” type, its dimension depends on the number of blocks for communication specified for the system. The “PIFPTable” is an array of “FlowMode” type, its dimension, one more time, depends on the number of partitions specified for the system. Finally, the “PartitionsExecutionSequenceTable” is an array of “PartitionTypes.PartitionNumber” (this type represents the “ID” of one partition) and its dimension depends on the value of “DefaultValues.PartitionsExecutionNumber” specified for the system.

**HardwareTypes**

In this package we have the representation of the physical memory which contains a description of the state of the system memory. The memory is divided into blocks and each block contains the starting address, the size and its state, which can be either free or occupied.

**ErrorTypes**

This package contains the types of errors and the facilities necessary for the SEF package. It has the operations summarized in Table 5.6:

<table>
<thead>
<tr>
<th>Ada Type</th>
<th>Type Classification</th>
<th>Z Type</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faults</td>
<td>enumeration</td>
<td>-</td>
<td>SEF#(01, 02), SYT#02</td>
</tr>
<tr>
<td>State</td>
<td>enumeration</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ErrorsTable</td>
<td>array</td>
<td>-</td>
<td>SEF#02, SYT#02</td>
</tr>
</tbody>
</table>

Table 5.6: ErrorTypes package.

The “Faults” type represents the different sorts of errors, which can take the following values:
Chapter 5. Implementation

- Hardware: for problems with memory;
- Configuration: representing a bad initialization of the system;
- Deadline: when the change of partitions exceed the time expected;
- Application: error assigned to a partition.

The “State” type represents two values, “NoError” or “Error” and the “ErrorsTable” type is an array with all of errors (“Faults”) and the state (“State”) corresponding to each of them.

SEF

In this package we have the “Errors/Table” table and the operation that changes its state, as shown in Table 5.7.

This package has only one operation that is used to change the status of errors in the “Errors/Table” table. It is in this table that the system saves information about the system errors and, when one of the shortcomings listed above happens, the state of the table is updated.

<table>
<thead>
<tr>
<th>Ada Operation</th>
<th>Operation Type</th>
<th>Z Operation</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>procedure</td>
<td>-</td>
<td>SEF#(01, 02)</td>
</tr>
</tbody>
</table>

Table 5.7: SEF package.

As we can see in the Code Listing 5.6, there exists an inherit annotation for the ErrorTypes package which is necessary for enabling the use of its types. As it is a variable package it contains one annotation to indicate that the package has a state (“Errors/Table”), the own annotation, and there is also the initializes annotation which requires the initialization of this state. In line 8, the “Errors/Table” is initialized. Then, in the next lines, the “Error” procedure to change the state of “Error/Table” is also annotated with a global, a derives and a proof annotations.
Chapter 5. Implementation

begin
with ErrorTypes;
--# inherit ErrorTypes;
package SEF
--# own Errors_Table;
--# initializes Errors_Table;

is

Errors_Table : ErrorTypes ErrorHandler :=
  ErrorTypesErrorHandler (ErrorTypes Faults => ErrorTypes.NoError);

procedure Error (Fault : in ErrorTypes Faults);
--# global in out Errors_Table;
--# derives Errors_Table from Errors_Table, Fault;
--# post Errors_Table(Fault) = ErrorTypes.Error;

end SEF;

Code Listing 5.6: SEF specification package

Hardware

This package deals with memory which, for our purposes, just needs a few operations listed below in Table 5.8.

<table>
<thead>
<tr>
<th>Ada Operation</th>
<th>Operation Type</th>
<th>Z Operation</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanAllocate</td>
<td>function</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AllocBlock</td>
<td>procedure</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Init</td>
<td>procedure</td>
<td>InitMemory</td>
<td>SEF#01</td>
</tr>
</tbody>
</table>

Table 5.8: Hardware package.

The specification, presented in Code Listing 5.6, shows three operations. The “CanAllocate” function returns a boolean value. This function checks if there exists space of memory free. The “AllocBlock” procedure really occupies a memory area and the “Init” procedure initializes the memory; this operation is carried out at the initialization of the system. This package was built with the refinement process, i.e. its state variable was decomposed thereafter in the body package. This refinement passes on to the body the proof annotations.
Chapter 5. Implementation

CMS

The CMS package’s objective is to configure the system and has the right procedures for its operation that are listed below in Table 5.9.

<table>
<thead>
<tr>
<th>Ada Operation</th>
<th>Operation Type</th>
<th>Z Operation</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitSystem</td>
<td>procedure</td>
<td>InitSystem</td>
<td>CMS#(01, 02, 03), SYT#(01, 03)</td>
</tr>
<tr>
<td>RunProcesses</td>
<td>procedure</td>
<td></td>
<td>PRT#(09, 10, 11), PRC#(05)</td>
</tr>
<tr>
<td>ContextSwitch</td>
<td>procedure</td>
<td>ContextSwitch</td>
<td>PRT#(07, 08)</td>
</tr>
</tbody>
</table>

Table 5.9: CMS package.

The “InitSystem” procedure, as the name suggests, serves to initialize the system. It initializes the memory, the kernel, the partitions, and the memory dedicated for communication.

The remaining procedures have the function of: 1) running the processes of one partition (the current partition); and 2) doing one context switch, i.e., switch the current partition respecting the pre-established sequence.
SYT

This package contains the system tables. It contains procedures for the initialization of their tables (partitions, communications and PIFP), and provides the facilities for managing these tables. The operations are summarized in Table 5.10.

<table>
<thead>
<tr>
<th>Ada Operation</th>
<th>Operation Type</th>
<th>Z Operation</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitPartitionsTable</td>
<td>procedure</td>
<td>Init</td>
<td>CMS#02, SYT#03</td>
</tr>
<tr>
<td>InitCommunicationsTable</td>
<td>procedure</td>
<td>InitCommunications</td>
<td>CMS#03, SYT#03</td>
</tr>
<tr>
<td>InitPIFPTable</td>
<td>procedure</td>
<td>InitPIFP</td>
<td>CMS#03, SYT#03</td>
</tr>
<tr>
<td>GetPartitionIndex</td>
<td>function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCommunicationList</td>
<td>procedure</td>
<td>UpdateComTable</td>
<td>PRC#(01, 02, 03), PIFP#06</td>
</tr>
<tr>
<td>Communication</td>
<td>procedure</td>
<td>Send, Receive</td>
<td>PRC#04, PIFP#04</td>
</tr>
<tr>
<td>DeleteComForPartition</td>
<td>procedure</td>
<td></td>
<td>PRC#(01, 02, 03)</td>
</tr>
<tr>
<td>ReadWrite</td>
<td>procedure</td>
<td>ReadWrite</td>
<td></td>
</tr>
<tr>
<td>CheckPIFP</td>
<td>procedure</td>
<td>CheckPIFP</td>
<td>PIFP#(02, 03, 05)</td>
</tr>
<tr>
<td>ChangePartitionMode</td>
<td>procedure</td>
<td>PartitionResumes,</td>
<td>PRT#03</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PartitionSuspends,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CurrPartitionError</td>
<td></td>
</tr>
<tr>
<td>GetDurationPRT</td>
<td>function</td>
<td>GetDuration</td>
<td>PRT#03</td>
</tr>
</tbody>
</table>

Table 5.10: SYT package.

The operations of initialization are used at the system boot time in the “InitSystem” procedure from CSM package. The “InitPartitionsTable” procedure ensures that partitions are not created with shared memory spaces; the “InitCommunicationsTable” procedure ensures that the blocks of memory for communication do not belong to any of the partitions.

The other operations are used in the normal system operating. The “UpdateCommunicationList” procedure is used by the processes with the aim to communicate with the processes in other partitions, the remaining operations are used by the kernel in the “ContextSwitch” procedure enabling or not the communications according to the partition information flow policy chosen for the system.

Each partition can only access its memory and the communication memory that respects the PIFP, they can not access the kernel memory or the other partitions memory, nor the communication memory that belongs to them. In the communication memory each
partition has access to the free blocks; the access to the occupied blocks is only possible when the target partition is the partition itself.

**ConfigValues**

This package is used to read the configuration file during the system booting. It contains local states which are used only in the startup of the system. The package is necessary to comply with the **SYT#03** (the initial state of the system should be a secure state) requirement. This package is extremely important because it verifies the integrity of the configuration file, checks whether the format is well defined, if there is enough memory to boot the system, and if there is a PIFP for all the partitions.

**File**

This package provides file utilities. It is distributed by praxis in the Tokeneer project [8].

**5.6 Other issues**

**The configuration file**

The configuration data is supplied in a file named `config.dat`. The format of a configuration data file is presented in Table 5.11. Each field is presented in a new line and takes the form of a field identifier followed by at least one space and the value of the field. Each line is terminated by a **CR** (carriage return) and **LF** (line feed). The file format has been selected to provide a user-friendly interface for entering values.

An example of a valid configuration file is presented in Appendix [D].
<table>
<thead>
<tr>
<th><strong>File Format</strong></th>
<th><strong>Comments</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>PartitionsNumber nn</td>
<td>Number of partitions.</td>
</tr>
<tr>
<td>NumberOfCommunicationsBlocks nn</td>
<td>Number of blocks for communication.</td>
</tr>
<tr>
<td>CommunicationBlocksSize nn</td>
<td>Size of communications blocks.</td>
</tr>
<tr>
<td>NumberOfProcessesPerPartition nn</td>
<td>Number of processes per partition.</td>
</tr>
<tr>
<td>PARTITION</td>
<td>Identifier of a partition description, must be followed by PartitionID, PartitionSIZE, PartitionDURATION and PROCESS (the process must be appear n times, specified above in NumberOfProcessesPerPartition).</td>
</tr>
<tr>
<td>PartitionID nn</td>
<td>The partition ID, the value range is 1 .. n defined above in PartitionsNumber. Repetitions are not allowed.</td>
</tr>
<tr>
<td>PartitionSIZE nn</td>
<td>The partition size.</td>
</tr>
<tr>
<td>PartitionDURATION nn</td>
<td>The partition duration.</td>
</tr>
<tr>
<td>PROCESS</td>
<td>Identifier of a process description. Must be following by ProcessOperation.</td>
</tr>
<tr>
<td>ProcessOperation tt</td>
<td>Values of tt are Communication and Nothing. If value is Communication must be following by CommunicationMode, CommunicationSize and CommunicationTo.</td>
</tr>
<tr>
<td>CommunicationMode tt</td>
<td>Values of tt are Read and Write.</td>
</tr>
<tr>
<td>CommunicationSize nn</td>
<td>Size of Communication. The value range is 1 .. NumberOfCommunicationsBlocks * CommunicationBlocksSize specified above.</td>
</tr>
<tr>
<td>CommunicationTo nn</td>
<td>Partition ID with which it is supposed to communicate with. The value range is 1 .. nn defined above in PartitionsNumber, with the exception of your own ID.</td>
</tr>
<tr>
<td>PartitionsExecutionNumber nn</td>
<td>Number of partitions that can be executed per cycle.</td>
</tr>
<tr>
<td>PartitionsExecutionSequence</td>
<td>Identifier of a partition sequence execution description. Must be followed by ID's of partitions, ordered. Repetitions are allowed.</td>
</tr>
<tr>
<td>PIFP</td>
<td>Identifier of a PIFP description. Must be followed by the description of PIFP.</td>
</tr>
<tr>
<td>ID</td>
<td>Must have all ID's of partitions and for each ID must be specified the policy of flow towards the other partitions. This is achieved by a value of PartitionID followed by the policy. This policy can be: N, R, W, and RW - respectively no-flow, read, write, and read-write.</td>
</tr>
</tbody>
</table>

**Table 5.11: Configuration file format.**
The communication mechanism

There are two modes of communication, one mode to read and an other mode to write. If the communication is “read”, then the process only reserves memory for future communication with a particular partition that will run later. This may seem like a failure, but if the partition to which the space was reserved in their respective space of time does not use it, then the kernel will automatically release the space of memory. One possible state of communication memory can be seen in Figure 5.4 which shows that the blocks are occupied according to the recipient of the message, that is, messages with different senders, but with the same recipient can be allocated in a unique block.

![Figure 5.4: A possible state of communication memory.](image)

The system tests

When using CbyC development method Altran Praxis undertakes tests against a fully detailed system specification using test coverage tools to ensure 100% coverage, and fully-automated testing to ease regression testing. In addition, in order to detect system integration failures they also test the full final system application. This procedure was not carried out though because it is beyond the scope of this MSc work.

The tests performed in this work were simple tests to verify if the behavior of the system was in accordance with what was expected. In the specification we used the model checking technique to observe the system behavior. Tests were done on the executable simulator that resulted from this work itself.
For these tests, we created a Ada package that allowed the visualization of the system state, i.e. allowed us to compare the system behavior and the state of the variable packages (packages with state) with the expected result. The Ada package created allowed us to see the state of the following variable packages: SYT.Partitions_Table, SYT_Communications_Table, SYT.PIFP_Table, SYT_PartitionsExecutionSequence_Table, SEF_Errors_Table, and Hardware_Mem.

Tests performed consisted of checking if it was the correct behavior, in accordance with the selected configuration file (config.dat), by viewing the state of the system. Various configuration files were created to test several possible scenarios. The different scenarios that were tested can be separated into two groups:

- **initialization**: that tests the correct start of the system, such as the memory allocation, the creation of partitions and processes, and the load of static tables (PIFP_Table and PartitionsExecutionSequence_Table);

- **execution**: that tests the normal behavior of the system, like the memory access control, the partitions exchange, the processes execution within a given partition, and the communications between partitions matching the established PIFP.

Overall, any tests that may be undertaken complement the static analysis of the system in order to confirm its dynamic behavior.

**Results**

The INFORMED stage is extremely useful because it allows us to have a mapping from the formal design to the code prior to writing the code.

The static analysis can start with only the specification packages. This lets us know if we adhere to the constraints of the SPARK language. With the specifications concluded, we move on to the development of the bodies and consequently the implementation of operations. With the facilities of the tools for SPARK we examine and verify the code as we go. The operations can be analyzed with the SPARK Examiner to verify the data and information flow properties of the code against the annotations in the specification. Thus, we obtain the check of code before it is compiled.

When the code of the packages is complete, the SPARK Examiner provides a check for run-time errors. This allows us to make sure that the code is free of errors that can cause
a run-time exception, such as accessing arrays outside their bounds or the overflow of numeric types.

SPARK Examiner generates a number of VCs (Verification Conditions) for each subprogram which needs to be shown true to conclude the proof of the subprogram. With the POGS we always have an overview of the validation of the project. When there are VCs that could not be discharged automatically, we can consult the file .siv of the subprogram and see the problem. With this method it was possible to find some basic flaws that had remained undetected. Figure 5.5 is an extract of the final result of POGS and, as we can see, the Examiner and the Simplifier were able to prove all VCs in a fully automatic way.

### Overall subprogram summary:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total subprograms</td>
<td>78</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fully proved</td>
<td></td>
<td>78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>with at least one</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>undischarged VC</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>with at least one</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>false VC</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>for which VCs have</td>
<td></td>
<td></td>
<td></td>
<td>78</td>
</tr>
<tr>
<td>been generated</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
VC summary:
```

Note: U/R denotes where the Simplifier has proved VCs using one or more user-defined proof rules.

```
Total VCs by type:
```

<table>
<thead>
<tr>
<th></th>
<th>Assert or Post</th>
<th>Precondition check</th>
<th>Check statement</th>
<th>Runtime check</th>
<th>Refinement VCs</th>
<th>Inheritance VCs</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Examiner</td>
<td>U/R</td>
<td>Total</td>
<td>Checker</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>----------------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>---------------</td>
<td>----------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>364</td>
<td>250</td>
<td>354</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>358</td>
<td>0</td>
<td>358</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 5.5: POGS summary.**
Chapter 6

Conclusions

This chapter concludes the work described in this dissertation. An overview of the main purpose of this study and its value to the scientific community is made. It is presented a comparison of the results achieved with the initial objectives proposed; some limitations that occurred during the work development also are pointed out. Finally, we present some recommendations for future work, both regarding possible extensions/improvements in the scope of this work or referring to new works with a similar development methodology.

6.1 Main aim and objectives

We can say that the main aim initially indicated, the use of CbyC development method to build a secure partitioning microkernel-based simulator, in order to confirm that it is indeed possible to develop systems able to achieve high levels of certification, was overall achieved.

As explained above, in Chapter 3, we outlined five objectives in order to achieve the main aim. The work undertaken and the results obtained in each step are summarized below.

1. **To investigate the system requirements:** this was indeed an essential step of the work. It enabled the collection of the necessary information in order to move
forward towards the system implementation goal. The documents SKPP [72] and ARINC-653 [137] were used as reference;

2. **To use the Z notation to create a high level specification:** this step was important for a full understanding of the features and properties of the system. It enabled the creation of one specification (formal model), with the requirements achieved in the previous step, in an unambiguous manner. In order to help the understanding of the features and properties of the system, we used an animator (ProZ) to obtain a more visual and interactive perspective of the specification. The resulting model is simple but very representative of the system core;

3. **To construct a design of the system with INFORMED process:** with the completion of this step, we produced an architecture that respects the specification created above. Although the formal specification has been started first, the final versions of the two stages were completed simultaneously. It allowed us, based on the information contained in the formal specification to develop an architecture, that besides helping in the elimination of unnecessary exchange of information, it also serves as a complementary component that facilitates the passage of the formal model for the packages used in the implementation of the system;

4. **To implement the system in SPARK:** the purpose of this step is to make the various packages of the system. It is divided into two stages: first, the specification creation; secondly, the bodies. The package specifications were created with the assistance of the formal model and the resulting architecture from the INFORMED design. These specifications include the Ada signature of the operations and the respective SPARK annotations. With this, we carried out a basic check - between the code and the annotations - to the information flow of operations. After the completion of the specification packages, their respective implementations (the body packages) were created. In the implementation, annotations that allowed the following checks were inserted: data flow analysis, information flow analysis, and proof of the absence of run-time errors. The properties of the formal specification were included in the implementation as SPARK proof annotations;

5. **To verify the system (using the SPARK Examiner toolset):** the SPARK Examiner tool was used over the resulting implementation from the previous step, in order to perform the verification that the properties described in the specifications were matched in the code. Thus, we have the possibility to make an early checking of the code (before compilation), so that we can prevent the introduction of some common mistakes, for example the use of uninitialized variables. We can
also check that we have no run-time errors, such as the access outside array or the overflow of numeric types. The SPARK approach reduces the need for testing, replacing it with analysis. The SPARK Examiner and SPADE Simplifier tools were used on the implementation source code, in order to discharge the associated proof obligations (Verification Conditions (VCs) generated). The VCs who have not been automatically proved were inspected, thereby finding some errors that may would have been unnoticed; also, some annotations were remade so that so they can be discharged automatically. In this way, we can reduce the number of VCs not discharged. However, some VCs were not automatically proved, these could/should be further analyzed and proved, by hand or with the aid of a semi-automatic prover, such as the Proof Checker tool.

The work reported in this dissertation addresses two subjects of great importance, as the Formal Methods and the Operating Systems. The first one is important to assure that a piece of software is well designed and implemented, and the second is also very important because is the core of any system - the services that run on it depend directly on its reliability. It is therefore inevitable, or at least it would be desirable, especially when it comes to critical systems, that the construction of OS were carried out with the use of formal methods. The work presented here, aims to highlight some aspects of this combination in order to contribute to a better understanding of the needs and benefits that may result from its use. This type of works, usually count on the association of a large team with high technical skills and many years of experience in both areas. It is clear enough that both of the subjects covered in this work have a high complexity; the case study (secure partitioning microkernel) that has very advanced OS concepts and the verification of software that becomes more complex and elaborate as the system size increases. It was assumed from the very beginning that the lack of experience would impose an important component of learning in the two fields in order to achieve the objective. However, despite the learning effort done, we are aware that a greater knowledge in these two areas is required to fully achieve the goals. There are two points, one in each area, that limited our work. In the FM area, we talk about the notion of manual proof; in the OS field, we talk about the integration with hardware. To some extent these aspects constituted a limitation, restricting the scope of our work. In spite of the difficulties above, it is expected that the results obtained may provide a basis for new approaches on the subject, enabling knowledge about the properties and structures necessary for its development.
Evaluating this work globally, it can be concluded that the various steps followed that compose the CbyC development method represent an added value when we want to build a system with high standards of reliability. As explained previously in Chapter 2, the fact that a software has been fully proved does not guarantee that it really does what it is expected to do. It may ensure that what ever is carried out is done without errors. However, the use of the CbyC development method allows one to visualize and understand, throughout the several steps, the system behavior, thus minimizing the risk above. Even if the software does not need to achieve high levels of certification, this methodology is very good to minimize mistakes and ultimately enables to make better software.

6.2 Contributions

The verification and certification of software is an area of enormous importance as it aims to ensure that no problems arise in delicate areas, safeguarding the preservation of human life and the environment that surrounds it. The subject under study is also of great relevance since it is a microkernel-based approach, which is clearly the central defining feature of an OS.

The final aim of this work was not only the system per se but also to better understand the development process and to increase the confidence on it. This work was part of a perspective to enable the increase of awareness and recognition that formal methods of software development can be used in a practical context. It is essentially a platform for knowledge transfer and use of advanced tools for software engineering in an important area such as safety critical.

The methodology used was based in the CbyC development method, which has been applied successfully in several contexts by Altran Praxis. As Anthony Hall says, “Correctness by Construction is a radical, effective and economical method of building software with high integrity especially for security-critical and safety-critical applications”. This methodology was used in the Tokeneer project that intended to demonstrate that it was possible to develop systems up to the level of rigor required by the highest standards of the Common Criteria.

Our motivation emerges from the same line of the Tokeneer project (with the differences of context and inherent limitations), but with another subject of study. The chosen case study was a secure partitioning microkernel, which is very important and absolutely
necessary in some critical systems. As previously mentioned, there are a considerable number of efforts towards the formalization and verification of kernels, nevertheless the majority does not concern this particular type of microkernel with partition and security properties. Only in the commercial field this kind of system exists with high levels of certification.

The contribution of this case study is therefore essentially a help to convince the software safety community that is really possible to develop secure systems in a rigorous way without neglecting the costs. Even so, we are aware that our work has not reached yet a state that complies to the level of rigor required by the standards of the highest levels of certification, which are required when security systems are concerned. Nevertheless, all the steps undertaken during our work and the fulfillment as match as possible of the CbyC development method allows us to assure that our secure partitioning microkernel-based simulator does indeed what it was meant to.

We can say that the use of the CbyC development method leads us to develop well built systems with a high level of quality. We strongly believe that this work may serve as a basis in the development of other new works in the regards to the verification field using FM.

6.3 Future work

Some suggestions for future contributions to improve the final result have emerged throughout this work. The main topics are presented below.

- to make a formal specification more complete, in order to encompass the whole system, in which all the properties of the system are proved, with more concrete elements for a direct mapping to implementation;

- to automate the passage of the properties of formal specification for the SPARK proof annotations, ensuring that the properties contained in the specification are rigorously passed on to the implementation;

- to add proof annotations to enable a full system coverage, and afterwards, if necessary, to prove all verification conditions generated by the SPARK tools, with the proof assistant or by hand, thereby ensuring that the implementation of the system is error-free:
to investigate the “concurrency part”\footnote{The “concurrency part” of SPARK, also known as RavenSPARK, was based on the Ravenscar Profile \cite{138} that defines a subset of the tasking features of Ada (giving deterministic concurrency) providing the means to construct highly-reliable concurrent programs.} of SPARK in order to add some concurrency properties, like time, since this work only used the “sequential part” of SPARK;

to change some aspects in order to make the system more flexible, like removing the restriction on the fixed number of processes per partition;

to add functionality to the kernel, such as support for keyboard or external devices, in order to make broader use of the system;

the followed step-by-step development method lead us to some choices, like specific methods used in the various steps, that can be replaced if considered more convenient (for example, changing the formal specification language, if people involved have more experience in another formal language).

As a final remark, it should be noted that, in order to help and guide in the choices and restrictions required by the hardware in works with a strong low level component, it is extremely useful that the team involved in it may count on members with high skills on OS. This is in fact an important factor both to guarantee the correct implementation in the hardware available, and to ensure thereafter a level of acceptable performance, given the wide range of peculiarities that can affect this.
Bibliography


Appendix A

Requirements

The list of the requirements is presented in this appendix. The documents SKPP [72] and ARINC-653 [137] were used as reference to this survey of requirements.

SKPP is a Common Criteria Protection Profile for separation kernels and ARINC-653 is a standard specification for system partitioning and scheduling, particularly in the avionics industry.

Similarities between SKPP and ARINC-653 are quite obvious. Both are used in applications requiring a design for high robust environments. The main difference is in communication. In SKPP communication between applications requires an authorization, whereas in ARINC-653 is not required any permission. Both SKPP and ARINC-653 provide a definition of mechanism that tries to address the enabling of multiple applications to be executed. The difference in SKPP is that each application should execute not only in a safe environment but also in a secure one.

Below are presented the functional requirements of the system. A functional requirement is a type of requirement that specifies a function that a system or component must be able to perform.

The Table [A.1] show the format in which the system requirements are presented.
### Requirements

**Identifier**
Is composed in the following way, Type#Number. Where Type parameter is the type of the requirement refers to, and Number parameter is a sequential number starting in one (when the type changes the counting restarts). The types allowed are:

- Configuration Management System (CMS)
- System Errors and Faults (SEF)
- System Tables (SYT)
- Partition (SYT)
- Process (PRC)
- Partition Information Flow Policy (PIFP)

| Identifier | Configuration Management System
|------------|--------------------------------|
| Title      | Configuration Management System
| Description| The system shall have a Configuration Management System (CMS)
| Rationale  | It is necessary to have a CMS with the ability to configure the system appropriated

Table A.2: CMS requirement # 01.
Appendix A. Requirements

<table>
<thead>
<tr>
<th>Identifier</th>
<th>CMS#02</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>System Information Requirements</td>
</tr>
<tr>
<td>Description</td>
<td>The system integrator shall have information about the system capabilities: memory and duration requirements</td>
</tr>
<tr>
<td>Rationale</td>
<td>It is necessary to system integrator to have the adequate information about the system. Without that information it will be impossible to provide the correct configuration</td>
</tr>
</tbody>
</table>

Table A.3: CMS requirement # 02.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>CMS#03</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Configuration Vector Information</td>
</tr>
</tbody>
</table>
| Description| The system shall have a configuration vector with the responsibility for:  
  - Memory allocation for each partition  
  - Creation of each partition  
  - For each partition the time processing  
  - Processes for each partition  
  - Communication principles  
  - Partition information flow policy |
| Rationale  | The configuration vector is responsibility of the system integrator. The information provided by the configuration vector is responsible for initialize the system with the appropriated information |

Table A.4: CMS requirement # 03.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>CMS#04</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Configuration Tables CMS</td>
</tr>
<tr>
<td>Description</td>
<td>Configuration tables shall be provided whit CMS</td>
</tr>
<tr>
<td>Rationale</td>
<td>Configuration tables must be build by the system integrator, they contain information that can be known when the configuration of partitions are defined</td>
</tr>
</tbody>
</table>

Table A.5: CMS requirement # 04.
Appendix A. Requirements

<table>
<thead>
<tr>
<th>Identifier</th>
<th>CMS#05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Configuration Tables Information</td>
</tr>
<tr>
<td>Description</td>
<td>Configuration tables shall be provided. Configuration tables shall have information about the communication, memory requirements and health monitor (HM)</td>
</tr>
<tr>
<td>Rationale</td>
<td>Configuration tables are required for the secure partitioning kernel work correctly. It provides the basic information for the system start working</td>
</tr>
</tbody>
</table>

Table A.6: CMS requirement # 05.

A.2 System error and faults requirements

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SEF#01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Faults Detection</td>
</tr>
<tr>
<td>Description</td>
<td>Faults shall be detected by several elements</td>
</tr>
<tr>
<td>Rationale</td>
<td>The elements that faults shall be detected are:</td>
</tr>
<tr>
<td></td>
<td>• Hardware: memory protection violation, privilege execution violation</td>
</tr>
<tr>
<td></td>
<td>• Configuration: initialization of the system fails</td>
</tr>
<tr>
<td></td>
<td>• Deadline: a partition exceed the time expected</td>
</tr>
<tr>
<td></td>
<td>• Application: error assigned to a partition</td>
</tr>
</tbody>
</table>

Table A.7: SEF requirement # 01.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SEF#02</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Error Report</td>
</tr>
<tr>
<td>Description</td>
<td>The HM shall be able to audit and report all the errors defined in the faults list</td>
</tr>
<tr>
<td>Rationale</td>
<td>The faults list has defined in the previous requirement</td>
</tr>
</tbody>
</table>

Table A.8: SEF requirement # 02.
A.3 System table requirements

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SYT#01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Health Monitor Composition</td>
</tr>
<tr>
<td>Description</td>
<td>The HM shall use configuration tables to handle each occurring error</td>
</tr>
<tr>
<td>Rationale</td>
<td>The HM table is provided by the system integrator in the initialization of the system by the CMS</td>
</tr>
</tbody>
</table>

Table A.9: SYT requirement # 01.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SYT#02</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>System Health Monitor Tables Level</td>
</tr>
<tr>
<td>Description</td>
<td>The system HM table shall define the level of an error (hardware, configuration, deadline, and application) according to the detected error and the state of the system</td>
</tr>
<tr>
<td>Rationale</td>
<td>System HM table gives the level of the error associated with the error and state of the system</td>
</tr>
</tbody>
</table>

Table A.10: SYT requirement # 02.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SYT#03</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Secure State Definition</td>
</tr>
<tr>
<td>Description</td>
<td>The initial state of the system shall be a secure state</td>
</tr>
<tr>
<td>Rationale</td>
<td>A secure state is a state where the system passes all the previous test functions</td>
</tr>
</tbody>
</table>

Table A.11: SYT requirement # 03.
## A.4 Partition requirements

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRT#01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Partition Attributes</td>
</tr>
<tr>
<td>Description</td>
<td>Each partition shall have the following attributes:</td>
</tr>
<tr>
<td></td>
<td>• Identifier: uniquely defined on a system-wide basis, and used to facilitate partition activation and message routing</td>
</tr>
<tr>
<td></td>
<td>• Memory requirements: defines memory bounds (minimum and maximum quotas) of the partition, with appropriate code/data segregation</td>
</tr>
<tr>
<td></td>
<td>• Duration: the amount of processor time given to the partition</td>
</tr>
<tr>
<td>Rationale</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table A.12: PRT requirement # 01.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRT#02</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Partition Resources Configuration</td>
</tr>
<tr>
<td>Description</td>
<td>The resources used by each configuration shall be specified at build time because of static configuration</td>
</tr>
<tr>
<td>Rationale</td>
<td>Static configuration change is the only permitted configuration. In this type of configuration partitions and processes are only created during the start-up of the system</td>
</tr>
</tbody>
</table>

Table A.13: PRT requirement # 02.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRT#03</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Partition Minimal Services</td>
</tr>
<tr>
<td>Description</td>
<td>The system shall have at least two services, SET_PARTITION_MODE and GET_PARTITION_MODE</td>
</tr>
<tr>
<td>Rationale</td>
<td>These two services will help to deal with the state of the partitions. SET_PARTITION_MODE request a change in its operational mode. GET_PARTITION_MODE returns the current mode of the partition</td>
</tr>
</tbody>
</table>

Table A.14: PRT requirement # 03.
## Appendix A. Requirements

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRT#04</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Partition Modes</td>
</tr>
<tr>
<td>Description</td>
<td>The Partition shall have the following modes:</td>
</tr>
<tr>
<td></td>
<td>• IDLE: in this mode the partition is not executing any processes within its allocated partition windows. The partition is not initialized (e.g., none of the ports associated to the partition are initialized), no processes are executing, but the time windows allocated to the partition are unchanged</td>
</tr>
<tr>
<td></td>
<td>• NORMAL: in this mode the process scheduler is active. All processes have been created and those that are in the ready state are able to run. The system is in an operational mode</td>
</tr>
<tr>
<td></td>
<td>• ERROR: in this mode the partition is not initialized again because an error has occurred.</td>
</tr>
<tr>
<td>Rationale</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Table A.15:** PRT requirement # 04.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRT#05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Partition Independency</td>
</tr>
<tr>
<td>Description</td>
<td>The operation mode of one partition shall be independent from others partitions</td>
</tr>
<tr>
<td>Rationale</td>
<td>This requirement is important to guarantee the space partitioning. If the mode of one partition is independent from others partitions then if one error occurs in one partition the others should be independent, and continuous working correctly</td>
</tr>
</tbody>
</table>

**Table A.16:** PRT requirement # 05.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRT#06</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Partition Scheduling in Partitions</td>
</tr>
<tr>
<td>Description</td>
<td>The schedule shall be fixed for the particular configuration of partitions on the processor</td>
</tr>
<tr>
<td>Rationale</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Table A.17:** PRT requirement # 06.
### PRT requirement # 07

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRT#07</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Partition Scheduling Characteristics</td>
</tr>
</tbody>
</table>
| Description | The main characteristics of the partition scheduling model shall be:  
  - The scheduling unit is a partition  
  - Partitions have no priority  
  - The scheduling algorithm is predetermined, repetitive with a fixed periodicity, and is configurable by the system configuration only. At least one partition window is allocated to each partition during each cycle |
| Rationale   | N/A |

Table A.18: PRT requirement # 07.

### PRT requirement # 08

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRT#08</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Partition Scheduling Mode Transitions</td>
</tr>
</tbody>
</table>
| Description | The possible mode transitions shall be:  
  - IDLE to NORMAL  
  - NORMAL to IDLE  
  - NORMAL to ERROR |
| Rationale   | N/A |

Table A.19: PRT requirement # 08.

### PRT requirement # 09

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRT#09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Partition Multiple Processes</td>
</tr>
<tr>
<td>Description</td>
<td>Multiple processes shall be supported within the partition</td>
</tr>
<tr>
<td>Rationale</td>
<td>Each partition could have multiple processes working inside the partition</td>
</tr>
</tbody>
</table>

Table A.20: PRT requirement # 09.
## Appendix A. Requirements

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRT#10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Partition Process Isolation</td>
</tr>
<tr>
<td>Description</td>
<td>The partition shall be responsible for the behavior of its internal processes</td>
</tr>
<tr>
<td>Rationale</td>
<td>The behavior of one process inside one partition could not affect the correct functionality of other processes in other partition</td>
</tr>
</tbody>
</table>

Table A.21: PRT requirement # 10.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRT#11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Partition Code</td>
</tr>
<tr>
<td>Description</td>
<td>Partition code shall execute only in user mode</td>
</tr>
<tr>
<td>Rationale</td>
<td>System has two modes of working, user mode and privilege mode. In case of partition code, it should execute in user mode</td>
</tr>
</tbody>
</table>

Table A.22: PRT requirement # 11.

### A.5 Process requirements

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRC#01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Processes Allocation</td>
</tr>
<tr>
<td>Description</td>
<td>Processes shall be created and allocated at partition initialization</td>
</tr>
<tr>
<td>Rationale</td>
<td>Processes are statically defined at build time</td>
</tr>
</tbody>
</table>

Table A.23: PRC requirement # 01.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRC#02</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Process Creation Unique Time</td>
</tr>
<tr>
<td>Description</td>
<td>Each process shall only be created once during life of the partition</td>
</tr>
<tr>
<td>Rationale</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table A.24: PRC requirement # 02.
### Appendix A. Requirements

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRC#03</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Processes in Partition Behavior</td>
</tr>
<tr>
<td>Description</td>
<td>A partition shall be able to reinitialize any of its processes at anytime, and shall also be able to prevent a process from becoming eligible to receive processor resources</td>
</tr>
<tr>
<td>Rationale</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table A.25: PRC requirement # 03.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRC#04</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Process Communication Mechanism</td>
</tr>
<tr>
<td>Description</td>
<td>The mechanisms used by the processes, for inter process communication and synchronization, are also created during the initialization phase, and are not destroyed</td>
</tr>
<tr>
<td>Rationale</td>
<td>Mechanisms for inter process communication and synchronization are created at initialization phase. This mechanism has an information flow authorization associated</td>
</tr>
</tbody>
</table>

Table A.26: PRC requirement # 04.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PRC#05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Processes Scheduling Model</td>
</tr>
<tr>
<td>Description</td>
<td>The main characteristics of the scheduling model used at the partition level shall be:</td>
</tr>
<tr>
<td></td>
<td>• One of the main activities of the OS is to arbitrate the competition that results in a partition when several processes of the partition each want exclusive control over the processor</td>
</tr>
<tr>
<td></td>
<td>• Each process has a priority</td>
</tr>
<tr>
<td></td>
<td>• The scheduling algorithm is priority preemptive. If several processes have the same current priority, the OS selects the oldest one</td>
</tr>
<tr>
<td></td>
<td>• Periodic and aperiodic scheduling of processes are both supported</td>
</tr>
<tr>
<td></td>
<td>• All the processes within a partition share the resources allocated to the partition</td>
</tr>
<tr>
<td>Rationale</td>
<td>Scheduler has two levels, the system level in which the scheduler sees only the partitions and not the processes inside the partition, and the partition level, in which the scheduler sees the processes inside the partition</td>
</tr>
</tbody>
</table>

Table A.27: PRC requirement # 05.
A.6 Partition information flow requirements

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PIFP#01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Partition Information Flow Policy</td>
</tr>
<tr>
<td>Description</td>
<td>The system shall have a partition information flow policy (PIFP)</td>
</tr>
<tr>
<td>Rationale</td>
<td>PIFP is one of the most important properties in the system. It implements the policy for processes from one partition communicate with processes from others partitions. It is statically defined and do not change while system is working. It should be implemented by the system integrator</td>
</tr>
</tbody>
</table>

Table A.28: PIFP requirement # 01.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PIFP#02</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Partition Information Flow Policy Enforcement</td>
</tr>
<tr>
<td>Description</td>
<td>The system shall enforce the PIFP for all possible operations that cause information flow between partitions</td>
</tr>
<tr>
<td>Rationale</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table A.29: PIFP requirement # 02.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PIFP#03</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Partition Information Flow Policy in Operations</td>
</tr>
<tr>
<td>Description</td>
<td>The system shall ensure that all operations that cause any information flow between partitions are covered by a partition information flow policy</td>
</tr>
<tr>
<td>Rationale</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table A.30: PIFP requirement # 03.
## Requirements

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PIFP#04</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Title</strong></td>
<td>Partition Information Flow Policy Transparency</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The data contents of a message are transparent to the message passing system</td>
</tr>
<tr>
<td><strong>Rationale</strong></td>
<td>The message passing system shall be independent from any partition and process. This is important to guarantee that when a process sends a message to another process, the message must remain confidential to other processes</td>
</tr>
</tbody>
</table>

**Table A.31: PIFP requirement # 04.**

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PIFP#05</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Title</strong></td>
<td>Partition Information Flow Policy OS Responsibility</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The OS is responsible for encapsulating and transporting messages, such that the message arrives at the destination unchanged</td>
</tr>
<tr>
<td><strong>Rationale</strong></td>
<td>Any fragmentation, segmentation, sequencing and routing of the message data by the core software, required to transport the data from source to destination, is invisible to the application(s). The core software is responsible for ensuring the integrity of the message data i.e., messages should not be corrupted in transmission</td>
</tr>
</tbody>
</table>

**Table A.32: PIFP requirement # 05.**

<table>
<thead>
<tr>
<th>Identifier</th>
<th>PIFP#06</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Title</strong></td>
<td>Partition Information Flow Policy Attributes</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The OS shall enforce the PIFP as a partition abstraction or a least privilege abstraction based on the flow(s) caused by an operation</td>
</tr>
<tr>
<td><strong>Rationale</strong></td>
<td>The principles of partition abstraction and least privilege abstraction have been defined. The principle associated to each partition and process depends on the type of communication. The principle should be defined by the system integrator</td>
</tr>
</tbody>
</table>

**Table A.33: PIFP requirement # 06.**
Appendix B

Z models

B.1 Types

The specification starts with the definition of the different types needed for the system. We separate them into four categories: memory, kernel, partition, and communication.

B.1.1 Memory

The allowed values for the memory addresses are defined in terms of a range. The lower bound ($NullAddr$) is the address with the value 0, and the upper bound ($MaxAddr$) is the value of the maximum address supported by the processor that will be used; in this case, it takes the value 2048.

\[ NullAddr, MaxAddr : \mathbb{N} \]
\[ NullAddr \equiv 0 \]
\[ MaxAddr \equiv 2048 \]

The $MemState$ type is defined below. Each block of memory has only one single state, out of the two possible, at any time.

\[ MemState ::= \text{FREE} \mid \text{OCCUPIED} \]

This type represents the “state” of a memory block, it represents a free block or an occupied block of memory. The next schema, $Block$, is the memory block definition.
Each *Block* of memory is constituted by a triplet:

- *Addr*: is the initial address of the block;
- *Sz*: defines the size of the block;
- *St*: represents the state of the block.

### B.1.2 Kernel

The processor can run in two distinct ways, with different privileges.

\[
WorkingMode ::= PrivilegeMode | UserMode
\]

The *PrivilegeMode* is meant to run the operations of the kernel and the *UserMode* is used for the partitions to run their operations. With the purpose to initialize the system, some constants were needed.

\[
\begin{align*}
\text{KernelSize} &= 200 \\
\text{PartitionsNumber} &= 3 \\
\text{PartitionSizeMinValue} &= 25 \\
\text{PartitionSizeMaxValue} &= 512 \\
\text{PartitionTimeMinValue} &= 1 \\
\text{PartitionTimeMaxValue} &= 5 \\
\text{CommunicationNumber} &= 3 \\
\text{CommunicationSize} &= 10
\end{align*}
\]

Below, we define each one of the constants above.

- *KernelSize*: specifies the space occupied by the kernel in memory;
- *PartitionsNumber*: represents the number of partitions in the system;
- *PartitionSizeMinValue*: represents the minimum size of a partition;
- *PartitionSizeMaxValue*: represents the maximum size of a partition;
- *PartitionTimeMinValue*: represents the minimum duration time of a partition;
Appendix B. \textit{Z models}

- \textit{PartitionTimeMaxValue}: represents the maximum duration time of a partition;
- \textit{CommunicationNumber}: specifies the number of communications supported by the system;
- \textit{CommunicationSize}: specifies the maximum size that each communication may have.

The \textit{ReturnCode} type identifies the state of the kernel, and is used to verify whether the kernel is in a normal state (\textit{OK}), in an error state (\textit{NOK}), or in a context switch state (\textit{CTX}).

\begin{equation}
\text{ReturnCode} ::= \text{OK} \mid \text{NOK} \mid \text{CTX}
\end{equation}

\textbf{B.1.3 Partition}

The \textit{State} type represents the state of a partition. Each partition can only assume one of the states below at any time.

\begin{equation}
\text{State} ::= \text{IDLE} \mid \text{NORMAL} \mid \text{ERROR}
\end{equation}

The names denote themselves the possible states:

- \textit{IDLE}: is the state of a partition that is ready to execute but not yet executing;
- \textit{NORMAL}: is the state of a partition that is currently executing;
- \textit{ERROR}: is the state of a partition that is not ready to execute, because an error has occurred.
The *Partition* type is, as the name indicates, the definition of a partition.

<table>
<thead>
<tr>
<th><strong>Partition</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ID</em>: N</td>
</tr>
<tr>
<td><em>Duration</em>: N</td>
</tr>
<tr>
<td><em>Addr</em>: N</td>
</tr>
<tr>
<td><em>Size</em>: N</td>
</tr>
<tr>
<td><em>Mode</em>: State</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><em>ID</em> &gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Duration</em> ∈ PartitionTimeMinValue . . PartitionTimeMaxValue</td>
</tr>
<tr>
<td><em>Addr</em> ∈ (KernelSize + CommunicationNumber * CommunicationSize) . . (MaxAddr – PartitionSizeMinValue)</td>
</tr>
<tr>
<td><em>Size</em> ∈ PartitionSizeMinValue . . PartitionSizeMaxValue</td>
</tr>
</tbody>
</table>

The *Partition* type is represented by a 5-tuple:

- *ID*: is the unique identifier of a partition;
- *Duration*: is the time available for the partition to run;
- *Addr*: is the initial address of memory for the partition;
- *Size*: is the size of memory available for the partition;
- *Mode*: represents the state of a partition.

### B.1.4 Communication

In order to implement the desired communication policy, the type *FlowMode* was created.

*FlowMode* ::= *N* | *R* | *W* | *RW*

Their values, as the characters suggest, represent the permission types. They are, respectively: “None”, “Read”, “Write”, and “Read/Write”. Communications within the same partition are of the “None” type, as it is an inter-partition communication system.
Appendix B. Z models

The Policy type is represented by a triplet and contains the allowed communication between partitions To and From.

```
Policy
To : N
From : N
Mode : FlowMode

To = From ⇒ Mode = N
```

Below is presented the schema Communication that represents a communication between two partitions. It contains the destiny and source partition IDs, the initial address in the memory, and the size of the communication.

```
Communication
To : N
From : N
Addr : N
Size : N

Size ∈ 0..CommunicationSize
```

The CommunicationOp type identifies the type of communication. It can be to Send or to Receive information.

```
CommunicationOp ::= Send | Receive
```

B.2 System state

The global state of the system is presented below by the Kernel schema.

```
Kernel
Status : ReturnCode
FirstADDRAvailable : N
FreeMemory : N
Memory : seq Block
Clock : N
Mode : WorkingMode
Partitions : P Partition
CurrPartition : N
PIFP : P Policy
Communications : P Communication
```
Appendix B. Z models

The kernel is represented by:

- **Status**: represents the state of the system, it is updated along the time by the operations;
- **FirstADDRAvailable**: assumes the first address of free memory;
- **FreeMemory**: assumes the size of free memory;
- **Memory**: is constituted by a sequence of memory blocks;
- **Clock**: is the representation of the elapsed time;
- **Mode**: shows the working mode allowed at the moment;
- **Partitions**: a set with the system partitions;
- **CurrPartition**: represents the running partition;
- **PIFP**: a set that contains the communications policies for all the system partitions;
- **Communications**: a set that contains the communications performed by the system.

This global system state is changed, along the time, by the operations executed. These operations were divided into four categories: 1) Initialization operations: that deal with the initialization of the system; 2) Partition operations: that comprise the operations of the partitions; 3) User operations: contain the operations that can run in UserMode; 4) Kernel operations: comprise the operations that can run in PrivilegeMode.
B.2.1 Initialization operations

The system initialization is divided into different parts. First, the system initializes with no partitions and with their sets empty, as shown in $Init$ schema.

<table>
<thead>
<tr>
<th>$Init$</th>
<th>$Kernel'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Status' = OK$</td>
<td>$FirstADDRAvailable' = NullAddr$</td>
</tr>
<tr>
<td>$FreeMemory' = NullAddr$</td>
<td>$Memory' = \langle \rangle$</td>
</tr>
<tr>
<td>$Clock' = 0$</td>
<td>$Mode' = PrivilegeMode$</td>
</tr>
<tr>
<td>$Partitions' = \emptyset$</td>
<td>$CurrPartition' = 0$</td>
</tr>
<tr>
<td>$PIFP' = \emptyset$</td>
<td>$Communications' = \emptyset$</td>
</tr>
</tbody>
</table>

Then, the system initializes the memory, as shown in $InitMemory$ schema. It builds a block with the size of available memory and allocates memory for the kernel.

<table>
<thead>
<tr>
<th>$InitMemory$</th>
<th>$\Delta Kernel$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FirstADDRAvailable = NullAddr$</td>
<td>$b1, b2 : Block$</td>
</tr>
<tr>
<td>$Mode = PrivilegeMode$</td>
<td>$b1.Addr = NullAddr$</td>
</tr>
<tr>
<td>$b1.Sz = KernelSize$</td>
<td>$b1.St = OCCUPIED$</td>
</tr>
<tr>
<td>$b2.Addr = b1.Sz$</td>
<td>$b2.St = FREE$</td>
</tr>
<tr>
<td>$b2.Sz = MaxAddr - b1.Sz$</td>
<td>$Status' = Status$</td>
</tr>
<tr>
<td>$FirstADDRAvailable' = b2.Addr$</td>
<td>$FreeMemory' = b2.Sz$</td>
</tr>
<tr>
<td>$Memory' = \langle b1 \rangle \setminus \langle b2 \rangle$</td>
<td>$Clock' = Clock$</td>
</tr>
<tr>
<td>$Mode' = Mode$</td>
<td>$Partitions' = Partitions$</td>
</tr>
<tr>
<td>$CurrPartition' = CurrPartition$</td>
<td>$PIFP' = PIFP$</td>
</tr>
<tr>
<td>$Communications' = Communications$</td>
<td></td>
</tr>
</tbody>
</table>
After that, the system communications are initialized by $\text{InitCommunications}$ schema with the defined size for communications.

\[
\begin{align*}
\text{InitCommunications} & \quad \Delta \text{Kernel} \\
& \quad b_1, b_2 : \text{Block} \\
& \quad c : \text{Communication} \\
& \quad \text{FirstADDRAvailable} \neq \text{NullAddr} \\
& \quad \text{Mode} = \text{PrivilegeMode} \\
& \quad \#\text{Communications} < \text{CommunicationNumber} \\
& \quad b_1.\text{Addr} = \text{FirstADDRAvailable} \\
& \quad b_1.\text{Sz} = \text{CommunicationSize} \\
& \quad b_1.\text{St} = \text{OCCUPIED} \\
& \quad b_2.\text{Addr} = b_1.\text{Addr} + b_1.\text{Sz} \\
& \quad b_2.\text{Sz} = \text{FreeMemory} - b_1.\text{Sz} \\
& \quad b_2.\text{St} = \text{FREE} \\
& \quad c.\text{To} = 0 \\
& \quad c.\text{From} = 0 \\
& \quad c.\text{Addr} = b_1.\text{Addr} \\
& \quad c.\text{Size} = 0 \\
& \quad \text{Status}' = \text{Status} \\
& \quad \text{FirstADDRAvailable}' = b_2.\text{Addr} \\
& \quad \text{FreeMemory}' = b_2.\text{Sz} \\
& \quad \text{Memory}' = \text{Front Memory} \cup \langle b_1 \rangle \cup \langle b_2 \rangle \\
& \quad \text{Clock}' = \text{Clock} \\
& \quad \text{Mode}' = \text{Mode} \\
& \quad \text{Partitions}' = \text{Partitions} \\
& \quad \text{CurrPartition}' = \text{CurrPartition} \\
& \quad \text{PIFP}' = \text{PIFP} \\
& \quad \text{Communications}' = \text{Communications} \cup \{ c \}
\end{align*}
\]
Finally, the PIFP is initialized by \textit{InitPIFP} schema considering the settled permissions for the system partitions.

\[
\begin{array}{|c|}
\hline
\text{InitPIFP} \\
\Delta \text{Kernel} \\
p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9 : \text{Policy} \\
\hline
\end{array}
\]

\[
\begin{array}{|l|}
\hline
\text{FirstADDRAvailable} \neq \text{NullAddr} \\
\text{Mode} = \text{PrivilegeMode} \\
\text{PIFP} = \emptyset \\
\# \text{Communications} = \text{CommunicationNumber} \\
\forall t_1, t_2 : \text{PIFP} \bullet t_1.\text{To} = t_2.\text{To} \Rightarrow t_1.\text{From} \neq t_2.\text{From} \\
p_1.\text{To} = 1 \land p_1.\text{From} = 1 \land p_1.\text{Mode} = N \\
p_2.\text{To} = 1 \land p_2.\text{From} = 2 \land p_2.\text{Mode} = N \\
p_3.\text{To} = 1 \land p_3.\text{From} = 3 \land p_3.\text{Mode} = RW \\
p_4.\text{To} = 2 \land p_4.\text{From} = 1 \land p_4.\text{Mode} = W \\
p_5.\text{To} = 2 \land p_5.\text{From} = 2 \land p_5.\text{Mode} = N \\
p_6.\text{To} = 2 \land p_6.\text{From} = 3 \land p_6.\text{Mode} = R \\
p_7.\text{To} = 3 \land p_7.\text{From} = 1 \land p_7.\text{Mode} = R \\
p_8.\text{To} = 3 \land p_8.\text{From} = 2 \land p_8.\text{Mode} = RW \\
p_9.\text{To} = 3 \land p_9.\text{From} = 3 \land p_9.\text{Mode} = N \\
\text{Status}' = \text{Status} \\
\text{FirstADDRAvailable}' = \text{FirstADDRAvailable} \\
\text{FreeMemory}' = \text{FreeMemory} \\
\text{Memory}' = \text{Memory} \\
\text{Clock}' = \text{Clock} \\
\text{Mode}' = \text{Mode} \\
\text{Partitions}' = \text{Partitions} \\
\text{CurrPartition}' = \text{CurrPartition} \\
\text{PIFP}' = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9\} \\
\text{Communications}' = \text{Communications} \\
\hline
\end{array}
\]

The initialization process is represented below by the \textit{InitSystem} operation. It is the combination of all the initializations described above.

\[
\text{InitSystem} \equiv \text{Init} ; \text{InitMemory} ; \text{InitCommunications} ; \text{InitPIFP}
\]
B.2.2 Partiton operations

The NewPartition schema creates a new partition in the system. For this purpose, it receives the ID ($id?$), the duration ($d?$), and the size ($s?$) as parameters.

\[
\text{NewPartition} \\
\Delta \text{Kernel} \\
id?, d?, s? : N \\
b1, b2 : Block \\
p : \text{Partition}
\]

\[
\begin{align*}
\text{Mode} &= \text{PrivilegeMode} \\
\#\text{Partitions} &< \text{PartitionsNumber} \\
\text{PIFP} &\neq \emptyset \\
\forall \ pr : \text{Partition} \mid \text{pr} \in \text{Partitions} \implies \text{pr.ID} \neq id? \\
id? &\in 1..\text{PartitionsNumber} \\
d? &\in \text{PartitionTimeMinValue}..\text{PartitionTimeMaxValue} \\
s? &\in \text{PartitionSizeMinValue}..\text{PartitionSizeMaxValue} \\
s? &\in 1..\text{FreeMemory} \\
b1.\text{Addr} &= \text{FirstADDRAvailable} \\
b1.\text{Sz} &= s? \\
b1.\text{St} &= \text{OCCUPIED} \\
b2.\text{Addr} &= \text{FirstADDRAvailable} + b1.\text{Sz} \\
b2.\text{Sz} &= \text{FreeMemory} - b1.\text{Sz} \\
b2.\text{St} &= \text{FREE} \\
p &\notin \text{Partitions} \\
p.\text{ID} &= id? \\
p.\text{Duration} &= d? \\
p.\text{Addr} &= \text{FirstADDRAvailable} \\
p.\text{Size} &= s? \\
p.\text{Mode} &= \text{IDLE} \\
\text{Status}' &= \text{Status} \\
\text{FirstADDRAvailable}' &= b2.\text{Addr} \\
\text{FreeMemory}' &= b2.\text{Sz} \\
\text{Memory}' &= \text{front Memory} \wedge \langle b1 \rangle \wedge \langle b2 \rangle \\
\text{Clock}' &= \text{Clock} \\
\text{Mode}' &= \text{Mode} \\
\text{Partitions}' &= \text{Partitions} \cup \{p\} \\
\text{CurrPartition}' &= \text{CurrPartition} \\
\text{PIFP}' &= \text{PIFP} \\
\text{Communications}' &= \text{Communications}
\end{align*}
\]
Appendix B. Z models

In PartitionResumes schema, a partition that is ready (the CurrPartition) enters the critical section and is then activated - its mode is changed from IDLE to NORMAL. The partition can only become active if no other partition is active.

\[
\begin{align*}
\text{PartitionResumes} & \quad \Delta \text{Kernel} \\
p1, p2 : \text{Partition} & \quad \text{Status} = \text{OK} \\
& \quad \text{Mode} = \text{PrivilegeMode} \\
& \quad \#\text{Partitions} = \text{PartitionsNumber} \\
& \quad \neg (\exists p : \text{Partition} \mid p \in \text{Partitions} \cdot p.\text{Mode} = \text{NORMAL}) \\
& \quad p1 \in \text{Partitions} \\
& \quad p1.\text{ID} = \text{CurrPartition} \\
& \quad p1.\text{Mode} = \text{IDLE} \\
& \quad p2.\text{ID} = p1.\text{ID} \\
& \quad p2.\text{Duration} = p1.\text{Duration} \\
& \quad p2.\text{Addr} = p1.\text{Addr} \\
& \quad p2.\text{Size} = p1.\text{Size} \\
& \quad p2.\text{Mode} = \text{NORMAL} \\
& \quad \text{Status'} = \text{Status} \\
& \quad \text{FirstADDRAvailable'} = \text{FirstADDRAvailable} \\
& \quad \text{FreeMemory'} = \text{FreeMemory} \\
& \quad \text{Memory'} = \text{Memory} \\
& \quad \text{Clock'} = \text{Clock} \\
& \quad \text{Mode'} = \text{UserMode} \\
& \quad \text{Partitions'} = (\text{Partitions} \setminus \{p1\}) \cup \{p2\} \\
& \quad \text{CurrPartition'} = \text{CurrPartition} \\
& \quad \text{PIFP'} = \text{PIFP} \\
& \quad \text{Communications'} = \text{Communications}
\end{align*}
\]
In *PartitionSuspends* schema, the active partition (the *CurrPartition*) leaves the critical section - its mode is changed from *NORMAL* to *IDLE*. This operation is used when the partition time slice is over. The system state is changed to *CTX*, in order to represent a context switch.

\[
\begin{align*}
\text{*PartitionSuspends*} \\
\Delta \text{Kernel} \\
p_1, p_2 : \text{Partition}
\end{align*}
\]

\[
\begin{align*}
\text{Status} = \text{OK} \\
\text{Mode} = \text{PrivilegeMode} \\
\#\text{Partitions} = \text{PartitionsNumber} \\
p_1 \in \text{Partitions} \\
p_1.\text{ID} = \text{CurrPartition} \\
p_1.\text{Mode} = \text{NORMAL} \\
p_1.\text{Duration} = \text{Clock} \\
p_2.\text{ID} = p_1.\text{ID} \\
p_2.\text{Duration} = p_1.\text{Duration} \\
p_2.\text{Addr} = p_1.\text{Addr} \\
p_2.\text{Size} = p_1.\text{Size} \\
p_2.\text{Mode} = \text{IDLE} \\
\text{Status}' = \text{CTX} \\
\text{FirstADDRAvailable}' = \text{FirstADDRAvailable} \\
\text{FreeMemory}' = \text{FreeMemory} \\
\text{Memory}' = \text{Memory} \\
\text{Clock}' = \text{Clock} \\
\text{Mode}' = \text{Mode} \\
\text{Partitions}' = (\text{Partitions} \setminus \{p_1\}) \cup \{p_2\} \\
\text{CurrPartition}' = \text{CurrPartition} \\
\text{PIFP}' = \text{PIFP} \\
\text{Communications}' = \text{Communications}
\end{align*}
\]
Appendix B. Z models

The \textit{CurrPartitionError} schema is an operation that is performed when the kernel is in \textit{NOK} state. It puts the \textit{CurrPartition} in the \textit{ERROR} mode, so that this partition can not be rescheduled again. As in the previous operation, the system state is changed to \textit{CTX}, in order to represent a context switch.

\begin{align*}
\text{\textit{CurrPartitionError}} \quad \Delta \text{Kernel} \\
p1, p2 : \text{Partition} \\
\text{Status} = \text{NOK} \\
\text{Mode} = \text{PrivilegeMode} \\
\#\text{Partitions} = \text{PartitionsNumber} \\
\text{CurrPartition} > 0 \\
\exists p : \text{Partition} \mid p \in \text{Partitions} \land p.\text{Mode} = \text{NORMAL} \\
p1 \in \text{Partitions} \\
p1.\text{ID} = \text{CurrPartition} \\
p2.\text{ID} = p1.\text{ID} \\
p2.\text{Duration} = p1.\text{Duration} \\
p2.\text{Addr} = p1.\text{Addr} \\
p2.\text{Size} = p1.\text{Size} \\
p2.\text{Mode} = \text{ERROR} \\
\text{Status} = \text{CTX} \\
\text{FirstADDRAvailable} = \text{FirstADDRAvailable} \\
\text{FreeMemory} = \text{FreeMemory} \\
\text{Memory} = \text{Memory} \\
\text{Clock} = p1.\text{Duration} \\
\text{Mode} = \text{Mode} \\
\text{Partitions} = (\text{Partitions} \setminus \{p1\}) \cup \{p2\} \\
\text{CurrPartition} = p2 \\
\text{PIFP} = \text{PIFP} \\
\text{Communications} = \text{Communications}
\end{align*}
The *GetDuration* schema, does not change the system state. This is a read operation and it returns the allowed duration (the time slice) of a given partition.

\[
\begin{array}{|c|}
\hline
\text{GetDuration} \\
\Xi Kernel \\
\text{id}?: \mathbb{N} \\
\text{time}!: \mathbb{N} \\
p : \text{Partition} \\
\hline
\text{Status} = \text{OK} \\
\text{Mode} = \text{PrivilegeMode} \\
\# Partitions = \text{PartitionsNumber} \\
\text{time}! = p.\text{Duration} \\
p \in \text{Partitions} \\
p.ID = \text{id}! \\
\end{array}
\]
B.2.3 User operations

For the purpose of this work, both the read and write operations were implemented as a single operation, \textit{ReadWrite}. This was because the concern is only on whether the operation can access a particular memory space. The \textit{ReadWrite} operation receives, as parameters, the address ($a$) and the size ($s$) of memory that it wants to access. If the desired memory space belongs to the \textit{CurrPartition}, the system state remains \textit{OK}, otherwise, it is changed to \textit{NOK}.

\begin{verbatim}
ReadWrite
\Delta Kernel
a? : N
s? : N
p : Partition

Status = OK
Mode = UserMode
s? > 0
p \in Partitions
p.ID = CurrPartition
p.Mode = NORMAL
Status' = if (p.Addr \leq a? \land p.Addr + p.Size \geq a? + s?)
    then (OK) else (NOK)
FirstADDRAvailable' = FirstADDRAvailable
FreeMemory' = FreeMemory
Memory' = Memory
Clock' = Clock
Mode' = Mode
Partitions' = Partitions
CurrPartition' = CurrPartition
PIFP' = PIFP
Communications' = Communications
\end{verbatim}
Appendix B. $Z$ models

The CheckPIFP operation checks whether a given communication is allowed by the PIFP table, i.e., it meets the PIFP policy of the system. The operation verifies if there is any permission for the communication between the $CurrPartition$ and the partition with the ID specified ($id?$). If so, the system state remains $OK$, it is changed to $NOK$ otherwise. If $op?$ equals to $Send$, then the policy mode can be $RW$ or $W$; on the other hand, if $op?$ equals to $Receive$, then the policy mode can be $RW$ or $R$.

<table>
<thead>
<tr>
<th>CheckPIFP</th>
<th>∆Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>$op?$ : CommunicationOp</td>
<td>$id?$ : N</td>
</tr>
<tr>
<td>$p : Policy$</td>
<td></td>
</tr>
</tbody>
</table>

- $Status = OK$
- $Mode = UserMode$
- $p \in PIFP$
- $p.To = \text{if } op? = Send \text{ then } id? \text{ else } CurrPartition$
- $p.From = \text{if } op? = Send \text{ then } CurrPartition \text{ else } id?$
- $Status' = \text{if } ((op? = Send \land (p.Mode = RW \lor p.Mode = W)) \lor (op? = Receive \land (p.Mode = RW \lor p.Mode = R)))$
- $\text{then } (OK) \text{ else } (NOK)$

- $FirstADDRAvailable' = FirstADDRAvailable$
- $FirstMemory' = FirstMemory$
- $Memory' = Memory$
- $Clock' = Clock$
- $Mode' = Mode$
- $Partitions' = Partitions$
- $CurrPartition' = CurrPartition$
- $PIFP' = PIFP$
- $Communications' = Communications$
Next, we present the communication operations meant to send or to receive a message, respectively *ComSend* and *ComReceive* schemas. The *ComSend* operation accepts two parameters, the ID of the partition to where the message is supposed to be sent (*to*?), and the size of the message (*s*?). If the communication memory has enough space to perform the send operation then the kernel state remains *OK*, it is changed to *NOK* otherwise.

\[
\begin{align*}
\text{ComSend} & \quad \Delta \text{Kernel} \\
\text{to}?: N & \quad s?: N \\
c1, c2: \text{Communication} & \\
\end{align*}
\]

\[
\begin{align*}
\text{Status} &= \text{OK} \\
\text{Mode} &= \text{UserMode} \\
\exists p: \text{Partition} \mid p \in \text{Partitions} & \quad p.ID = \text{to}? \\
\text{to}? & \neq \text{CurrPartition} \\
s? & > 0 \\
c1 & \in \text{Communications} \\
c1.To = 0 & \lor c1.To = \text{to}? \\
c2.To = \text{to}? & \\
c2.From = \text{CurrPartition} \\
c2.Addr = c1.Addr \\
c2.Size = \text{if } (c1.Size + s?) \leq \text{CommunicationSize} \text{ then } (c1.Size + s?) \text{ else } (0) \\
\text{Status}' &= \text{if } (c2.Size > 0) \text{ then } (\text{OK}) \text{ else } (\text{NOK}) \\
\text{FirstADDRAvailable}' &= \text{FirstADDRAvailable} \\
\text{FreeMemory}' &= \text{FreeMemory} \\
\text{Memory}' &= \text{Memory} \\
\text{Clock}' &= \text{Clock} \\
\text{Mode}' &= \text{Mode} \\
\text{Partitions}' &= \text{Partitions} \\
\text{CurrPartition}' &= \text{CurrPartition} \\
\text{PIFP}' &= \text{PIFP} \\
\text{Communications}' &= \text{if } (\text{Status}' = \text{OK}) \text{ then } ((\text{Communications} \setminus \{c1\}) \cup \{c2\}) \text{ else } (\text{Communications})
\end{align*}
\]
The `ComReceive` operation accepts one input parameter that represents the ID of the partition from where the message comes (from?). This operation returns the initial memory address of the message. However, if it tries to access a partition that was not included in the communication, then the kernel state is changed to `NOK`.

\[\begin{align*}
\text{ComReceive} \\
\Delta \text{Kernel} \\
\text{from?} : N \\
\alpha! : N \\
c : \text{Communication}
\end{align*}\]

\[\begin{align*}
\text{Status} &= OK \\
\text{Mode} &= \text{UserMode} \\
\text{CurrPartition} &\neq 0 \\
\alpha! &= \text{if} \ (\text{Status}' = OK) \ \text{then} \ (c.\text{Addr}) \ \text{else} \ (0) \\
c \in \text{Communications} \\
\text{Status}' &= \text{if} \ (c.\text{To} = \text{CurrPartition} \land c.\text{From} = \text{from?}) \ \text{then} \ (OK) \ \text{else} \ (NOK) \\
\text{FirstADDRAvailable}' &= \text{FirstADDRAvailable} \\
\text{FreeMemory}' &= \text{FreeMemory} \\
\text{Memory}' &= \text{Memory} \\
\text{Clock}' &= \text{Clock} \\
\text{Mode}' &= \text{Mode} \\
\text{Partitions}' &= \text{Partitions} \\
\text{CurrPartition}' &= \text{CurrPartition} \\
\text{PIFP}' &= \text{PIFP} \\
\text{Communications}' &= \text{Communications}
\end{align*}\]

The operations to perform communications require permission from PIFP policy to execute, therefore the operations can be summarized as below.

\[\begin{align*}
\text{Send}(\text{to?}, s?) &\equiv \text{CheckPIFP}(\text{Send}, \text{to?}) \ ; \ \text{ComSend}(\text{to?}, s?)
\end{align*}\]

and

\[\begin{align*}
\text{Receive}(\text{from?}, a!) &\equiv \text{CheckPIFP}(\text{Receive}, \text{from?}) \ ; \ \text{ComReceive}(\text{from?}, a!)
\end{align*}\]
B.2.4 Kernel operations

After the completion of the initialization, if all went well ($Status = OK$), the kernel starts with the schema presented below. This operation picks one of the partitions and assigns its ID to the $CurrPartition$.

\[
\begin{align*}
\text{StartKernel} \equiv & \text{Kernel} \\
\Delta Kernel \equiv & p : \text{Partition} \\
\text{Status} = & OK \\
\text{Mode} = & \text{PrivilegeMode} \\
\# \text{Partitions} = & \text{PartitionsNumber} \\
\text{CurrPartition} = & \text{NullAddr} \\
p \in & \text{Partitions} \\
\text{Status}' = & \text{Status} \\
\text{FirstADDRAvailable}' = & \text{FirstADDRAvailable} \\
\text{FreeMemory}' = & \text{FreeMemory} \\
\text{Memory}' = & \text{Memory} \\
\text{Clock}' = & \text{Clock} \\
\text{Mode}' = & \text{Mode} \\
\text{Partitions}' = & \text{Partitions} \\
\text{CurrPartition}' = & p.ID \\
\text{PIFP}' = & \text{PIFP} \\
\text{Communications}' = & \text{Communications}
\end{align*}
\]

In order to complete the step of starting the kernel, it is necessary to perform the $\text{PartitionResumes}$ operation. The complete step to start the kernel is shown below.

\[
\text{Start} \equiv \text{StartKernel} ; \text{PartitionResumes}
\]
The \textit{UpdateComTable} schema is utilized to update the communication table, deleting the entries that have both a \textit{To} partition that has just run, and a \textit{To} partition that has in \textit{ERROR} mode. This operation is performed in the context switch mechanism.

\[
\begin{align*}
\text{UpdateComTable} & \\
\Delta \text{Kernel} & \\
\text{p : Partition} & \\
\text{c1, c2 : Communication} & \\
\text{Status} = \text{CTX} & \\
\text{Mode} = \text{PrivilegeMode} & \\
\text{p} \in \text{Partitions} & \\
\text{p.Mode} = \text{ERROR} & \\
\text{c1} \in \text{Communications} & \\
\text{c1.To} = \text{CurrPartition} \lor \text{c1.To} = \text{p.ID} & \\
\text{c2.To} = 0 & \\
\text{c2.From} = 0 & \\
\text{c2.Addr} = \text{c1.Addr} & \\
\text{c2.Size} = \text{CommunicationSize} & \\
\text{Status}' = \text{Status} & \\
\text{FirstADDRAvailable}' = \text{FirstADDRAvailable} & \\
\text{FreeMemory}' = \text{FreeMemory} & \\
\text{Memory}' = \text{Memory} & \\
\text{Clock}' = \text{Clock} & \\
\text{Mode}' = \text{Mode} & \\
\text{Partitions}' = \text{Partitions} & \\
\text{CurrPartition}' = \text{CurrPartition} & \\
\text{PIFP}' = \text{PIFP} & \\
\text{Communications}' = (\text{Communications} \setminus \{c1\}) \cup \{c2\} &
\end{align*}
\]
Appendix B. Z models

The Scheduler schema resets the Clock to zero and changes the CurrPartition to one partition that is in IDLE mode. If there is not exist any partition in IDLE mode then the kernel state is changed to NOK.

\[
\begin{align*}
\text{Scheduler} & \\
\Delta \text{Kernel} & \\
p : \text{Partition} & \\
\text{Status} = \text{CTX} & \\
\text{Mode} = \text{PrivilegeMode} & \\
\text{Status}' = \text{if } (\exists \text{prt} : \text{Partition } \mid \text{prt} \in \text{Partitions} \bullet \text{prt.Mode} = \text{IDLE} \land \text{prt.ID} = p.ID) \\
\text{then } (\text{OK}) \text{ else } (\text{NOK}) & \\
\text{FirstADDRAvailable}' = \text{FirstADDRAvailable} & \\
\text{FreeMemory}' = \text{FreeMemory} & \\
\text{Memory}' = \text{Memory} & \\
\text{Clock}' = 0 & \\
\text{Mode}' = \text{Mode} & \\
\text{Partitions}' = \text{Partitions} & \\
\text{CurrPartition}' = \text{if } (\text{Status}' = \text{OK}) \text{ then } (p.ID) \text{ else } (0) & \\
\text{PIFP}' = \text{PIFP} & \\
\text{Communications}' = \text{Communications} & \\
\end{align*}
\]

This Scheduler schema is part of the ContextSwitch mechanism presented below.

\[
\text{ContextSwitch} \equiv (\text{PartitionSuspends } \lor \text{CurrPartitionError}) \triangledown \text{UpdateComTable} \triangledown \\
\text{Scheduler} \triangledown \text{PartitionResumes}
\]

In this operation, either the PartitionSuspends or the CurrPartitionError is performed according to the kernel state is OK or NOK respectively.
The *ChangeMode* operation modifies the kernel *Mode*, it is changed between *PrivilegeMode* and *UserMode*.

\[
\begin{array}{l}
\text{ChangeMode} \\
\Delta \text{Kernel} \\
\begin{array}{l}
\text{CurrPartition} \neq 0 \\
\text{Status}' = \text{Status} \\
\text{FirstADDRAvailable}' = \text{FirstADDRAvailable} \\
\text{FreeMemory}' = \text{FreeMemory} \\
\text{Memory}' = \text{Memory} \\
\text{Clock}' = \text{Clock} \\
\text{Mode}' = \text{if } (\text{Mode} = \text{PrivilegeMode}) \\
\quad \text{then } (\text{UserMode}) \text{ else } (\text{PrivilegeMode}) \\
\text{Partitions}' = \text{Partitions} \\
\text{CurrPartition}' = \text{CurrPartition} \\
\text{PIFP}' = \text{PIFP} \\
\text{Communications}' = \text{Communications}
\end{array}
\end{array}
\]

The *Tick* operation has the purpose to give a tick of the CPU clock (the kernel *Clock* is increased in one unit), an interrupt is made, and the kernel *Mode* is changed to the *PrivilegeMode*.

\[
\begin{array}{l}
\text{Tick} \\
\Delta \text{Kernel} \\
\begin{array}{l}
\text{Mode} = \text{UserMode} \\
\text{Status}' = \text{Status} \\
\text{FirstADDRAvailable}' = \text{FirstADDRAvailable} \\
\text{FreeMemory}' = \text{FreeMemory} \\
\text{Memory}' = \text{Memory} \\
\text{Clock}' = \text{Clock} + 1 \\
\text{Mode}' = \text{PrivilegeMode} \\
\text{Partitions}' = \text{Partitions} \\
\text{CurrPartition}' = \text{CurrPartition} \\
\text{PIFP}' = \text{PIFP} \\
\text{Communications}' = \text{Communications}
\end{array}
\end{array}
\]
Appendix C

SPARK packages

C.1 DefaultValues

```asciidoc
package DefaultValues is
  Name_Of_Configuration_File : constant String := "/configFile.dat";
  Memory_Size : constant := 2048;
  KernelSize : constant := 512;
  Number_Of_Partitions : constant := 5;
  PartitionSizeMinValue : constant := 100;
  PartitionSizeMaxValue : constant := 1024;
  PartitionMaxTime : constant := 3600;
  Number_Of_Processes_Per_Partition : constant := 2;
  Number_Of_Communications_Blocks : constant := 3;
  CommunicationsBlockSizeMinValue : constant := 1;
  CommunicationsBlockSizeMaxValue : constant := 30;
  PartitionsExecutionNumber : constant := 6;
end DefaultValues;
```

Code Listing C.1: DefaultValues package

C.2 PartitionTypes

```asciidoc
with DefaultValues;
--# inherit DefaultValues;
```
package PartitionTypes is
  type Partition_Mode is (IDLE, NORMAL, ERROR);
  subtype PartitionSizeAllowValues is Integer range DefaultValues.PartitionSizeMinValue .. DefaultValues.PartitionSizeMaxValue;
  subtype ReadWriteAllowValues is Integer range 1 .. DefaultValues.PartitionSizeMaxValue;
  subtype AddrAllowValues is Integer range 0 .. DefaultValues.Memory_Size;

type Tuple is record
  init : AddrAllowValues;
  sz : PartitionSizeAllowValues;
end record;

type Tuple2 is record
  init : AddrAllowValues;
  sz : ReadWriteAllowValues;
end record;

subtype Index_Range is Integer range 1 .. DefaultValues.Number_Of_Communications_Blocks;

type Communication_Mode is (FREE, READ, WRITE, BLOCKED);

type Communication is array (Index_Range) of Communication_Mode;

subtype IdType is Integer range 0 .. DefaultValues.Number_Of_Partitions;

subtype PartitionsNumber is Integer range 1 .. DefaultValues.Number_Of_Partitions;

type OperationType is (OP_Nothing, OP_Communication, OP_ReadWrite);

subtype CommunicationMode is Communication_Mode range READ .. WRITE;

subtype CommunicationsBlockSizeAllowValues is Integer range DefaultValues.CommunicationsBlockSizeMinValue .. DefaultValues.CommunicationsBlockSizeMaxValue;

subtype SizeOfCommunication is Integer range CommunicationsBlockSizeAllowValues’First .. DefaultValues.Number_Of_Communications_Blocks * + CommunicationsBlockSizeAllowValues’Last;

subtype SizeOfCommunication2 is Integer range 0 .. DefaultValues.Number_Of_Communications_Blocks * + CommunicationsBlockSizeAllowValues’Last;

type ProcessesType is record
  operation : OperationType;
  blk : Tuple2;
  mode : CommunicationMode;
  size : SizeOfCommunication;
  to : PartitionsNumber;
end record;
Appendix C. \textit{SPARK packages}

\subsection*{C.3 PRT}

\textbf{Specification}

\begin{verbatim}
with PartitionTypes;
--# inherit PartitionTypes;
package PRT is

  function GetID(P : PartitionTypes.Partition) return PartitionTypes.IdType;
--# return P.ID;

  function GetDuration(P : PartitionTypes.Partition) return PartitionTypes.Partition.Duration;
--# return P.Duration;

  function GetPartitionMode(P : PartitionTypes.Partition) return PartitionTypes.Partition.Mode;
--# return P.Mode;

  procedure SetPartitionMode(P : in out PartitionTypes.Partition;
      Mode_Partition : in);
--# derives P from P, Mode_Partition;
--# post P.ID = P'.ID and
--#   P.MemoryBounds = P'.MemoryBounds and
--#   P.Duration = P'.Duration and
--#   P.Mode = Mode_Partition and
--#   P.Processes = P'.Processes and
--#   P.Com = P'.Com;

  procedure Init(P : out PartitionTypes.Partition;

end PRT;
\end{verbatim}

\textbf{Code Listing C.2: PartitionTypes package}
Appendix C. SPARK packages

ID_Partition : in PartitionTypes.IdType;
MemoryBounds_Partition : in PartitionTypes.Tuple;
Duration_Partition : in PartitionTypes.Partition_Duration;
Mode_Partition : in PartitionTypes.Partition_Mode;
Proc : in PartitionTypes.Processes_List);

--- derives P from ID_Partition, MemoryBounds_Partition, Duration_Partition, Mode_Partition, Proc;
--- post P.ID = ID_Partition and
--- P.MemoryBounds = MemoryBounds_Partition and
--- P.Duration = Duration_Partition and
--- P.Mode = Mode_Partition and
--- P.Processes = Proc and
--- P.Com = PartitionTypes.Communication'(PartitionTypes.Index_Range ← PartitionTypes.BLOCKED);

procedure ClearCommunicationList(P : in out PartitionTypes.Partition);
--- derives P from P;
--- post P.ID = P>ID.ID and
--- P.MemoryBounds = P".MemoryBounds and
--- P.Duration = P".Duration and
--- P.Mode = P".Mode and
--- P.Processes = P".Processes and
--- P.Com = PartitionTypes.Communication'(PartitionTypes.Index_Range ← PartitionTypes.BLOCKED);

end PRT;

Code Listing C.3: PRT specification package

Body

package body PRT is

function GetID(P : PartitionTypes.Partition) return PartitionTypes.IdType is
begin
return P.ID;
end GetID;

function GetDuration(P : PartitionTypes.Partition) return ←
PartitionTypes.Partition.Duration is
begin
return P.Duration;
end GetDuration;

function GetPartitionMode(P : PartitionTypes.Partition) return ←
PartitionTypes.Partition.Mode is
begin
return P.Mode;
end GetPartitionMode;

procedure SetPartitionMode(P : in out PartitionTypes.Partition;
Mode_Partition : in ←
PartitionTypes.Partition.Mode) is
begin
  P.ID := P.ID;
  P.MemoryBounds := P.MemoryBounds;
  P.Duration := P.Duration;
  P.Mode := Mode_Parition;
  P.Com := P.Com;
  P.Processes := P.Processes;
end SetPartitionMode;

procedure Init(P: out PartitionTypes.Partition;
                ID_Parition: in PartitionTypes.IdType;
                MemoryBounds_Parition: in PartitionTypes.Tuple;
                Duration_Parition: in PartitionTypes.Partition_Duration;
                Mode_Parition: in PartitionTypes.Partition_Mode;
                Proc: in PartitionTypes.Processes_List) is
  begin
    P.ID := ID_Parition;
    P.MemoryBounds := MemoryBounds_Parition;
    P.Duration := Duration_Parition;
    P.Mode := Mode_Parition;
    P.Processes := Proc;
    P.Com := PartitionTypes.Communication'(PartitionTypes.Index_Range => PartitionTypes.BLOCKED);
end Init;

procedure ClearCommunicationList(P: in out PartitionTypes.Partition) is
  begin
    P.ID := P.ID;
    P.MemoryBounds := P.MemoryBounds;
    P.Duration := P.Duration;
    P.Mode := P.Mode;
    P.Com := PartitionTypes.Communication'(PartitionTypes.Index_Range => PartitionTypes.BLOCKED);
end ClearCommunicationList;

end PRT;

CODE LISTING C.4: PRT body package

C.4 TablesTypes

with DefaultValues, PartitionTypes;
--# inherit DefaultValues, PartitionTypes;
package TablesTypes is
  type PartitionsTable is array(PartitionTypes.PartitionsNumber) of PartitionTypes.Partition;

  type Blocks_State is (FREE, OCCUPIED);

  type Tuple_Communication is record
    init: PartitionTypes.AddrAllowValues;
  end record;
Appendix C. SPARK packages

```
sz: PartitionTypes.CommunicationsBlockSizeAllowValues;
end record;

type CommunicationState is record
  Blk: TupleCommunication;
  State: Blocks.State;
  From_Partition: PartitionTypes.IdType;
  To_Partition: PartitionTypes.PartitionsNumber;
  Mode: PartitionTypes.CommunicationMode;
  FreeSize: Natural;
end record;

subtype CommunicationsBlocksNumber is Integer range 1 ..
  DefaultValues.Number_Of_Communications_Blocks;

type CommunicationsTable is array (CommunicationsBlocksNumber) of CommunicationState;

type FlowMode is (R, W, RW, N);

type Partitions is array (PartitionTypes.PartitionsNumber) of FlowMode;

type PIFPTable is array (PartitionTypes.PartitionsNumber) of Partitions;

subtype PartitionsExecutionIndex is Integer range 1 ..
  DefaultValues.PartitionsExecutionNumber;

type PartitionsExecutionSequenceTable is array (PartitionsExecutionIndex) of PartitionTypes.PartitionsNumber;
```

C.5 HardwareTypes

```
with DefaultValues;
--# inherit DefaultValues;
package HardwareTypes is

  NullAddr: constant := 0;

  subtype Memory_Range is Integer range NullAddr .. DefaultValues.Memroy_Size;

  type State is (FREE, OCCUPIED);

type Block is record
  Addr: Memory_Range;
  Sz: Memory_Range;
  St: State;
end record;
```
type PhysicalMemory is array(Memory_Range) of Block;
end HardwareTypes;

Code Listing C.6: HardwareTypes package

C.6 ErrorTypes

package ErrorTypes is
  type Faults is (Hardware, Configuration, Deadline, Application);
  type State is (NoError, Error);
  type ErrorsTable is array(Faults) of State;
end ErrorTypes;

Code Listing C.7: ErrorTypes package

C.7 SEF

Specification

with ErrorTypes;
--# inherit ErrorTypes;
package SEF
--# own Errors_Table;
--# initializes Errors_Table;
is
  Errors_Table : ErrorTypes.ErrorsTable := ← ErrorTypes.ErrorsTable'((ErrorTypes.Faults => ErrorTypes.NoError);
  procedure Error (Fault : in ErrorTypes.Faults);
--# global in out Errors_Table;
--# derives Errors_Table from Errors_Table, Fault;
--# post Errors_Table(Fault) = ErrorTypes.Error;
end SEF;

Code Listing C.8: SEF specification package

Body

package body SEF is
  procedure Error (Fault : in ErrorTypes.Faults) is
begin
Appendix C. SPARC packages

C.8 Hardware

Specification

with HardwareTypes, DefaultValues;
use type HardwareTypes.State;
--# inherit HardwareTypes, DefaultValues;
package Hardware
--# own Mem;
--# initializes Mem;
is

function CanAllocate(S : HardwareTypes.MemoryRange) return Boolean;
--# global Mem;

procedure AllocBlock(S : in HardwareTypes.MemoryRange;
A : out HardwareTypes.MemoryRange;
OK : out Boolean);
--# global in out Mem;
--# derives Mem from Mem, S &
A from Mem, S &
--# OK from Mem, S &

procedure Init(Size : in Integer; Success : out Boolean);
--# global Mem;
--# derives Mem from Size &
Success from Size;

end Hardware;

Body

package body Hardware
--# own Mem is M, First_ADDR_Available, Free_Memory;
is

M : HardwareTypes.PhysicalMemory := ←
HardwareTypes.PhysicalMemory'(
HardwareTypes.MemoryRange => ←
HardwareTypes.Block'(Addr => HardwareTypes.MemoryRange'First, Sz => ←
HardwareTypes.MemoryRange'First, St => HardwareTypes.FREE));
First_ADDR_Available : HardwareTypes.Mem_Range := First;

Free_Memory : HardwareTypes.Mem_Range := first;

function CanAllocate(S : HardwareTypes.Mem_Range) return Boolean
--# global Free_Memory;
--# return Free_Memory >= S;
is
  r : Boolean;
begin
  r := False;
  if Free_Memory >= S then r := True; end if;
  return r;
end CanAllocate;

procedure AllocBlock(S : in HardwareTypes.Mem_Range;
A : out HardwareTypes.Mem_Range;
OK : out Boolean)
--# global in out Free_Memory, First_ADDR_Available, M;
--# derives M from M, S, Free_Memory, First_ADDR_Available &
--# A from M, First_ADDR_Available, S, Free_Memory &
--# First_ADDR_Available from First_ADDR_Available, S, <--
Free_Memory, M &
--# Free_Memory from Free_Memory, S, First_ADDR_Available, M &
--# OK from S, Free_Memory, First_ADDR_Available, M;
--# post ( OK -> ((A = M(First_ADDR_Available)).Addr) and
--# CanAllocate(S, Free_Memory) and
--# (Free_Memory = Free_Memory - S) and
--# (M(First_ADDR_Available).Sz = S) and
--# (M(First_ADDR_Available).St = HardwareTypes.OCCUPIED) and
--# (First_ADDR_Available = First_ADDR_Available + 1) and
--# (M(First_ADDR_Available).Addr = M(First_ADDR_Available).Addr + S) and
--# OK = True )
begin
  A := 0;
  OK := False;
  if (CanAllocate(S) and Free_Memory - S >= 0 and First_ADDR_Available <=
<= HardwareTypes.Mem_Range’Last - 1 and M(First_ADDR_Available).Addr <=
+ S <= HardwareTypes.Mem_Range’Last) then
    M(First_ADDR_Available).St := HardwareTypes.OCCUPIED;
    M(First_ADDR_Available).Sz := S;
    A := M(First_ADDR_Available).Addr;
    M(First_ADDR_Available + 1).Addr := M(First_ADDR_Available).Addr + S;
    First_ADDR_Available := First_ADDR_Available + 1;
    Free_Memory := Free_Memory - S;
    M(First_ADDR_Available).Sz := Free_Memory;
    OK := True;
  end if;
end AllocBlock;
procedure Init (Size : in Integer; Success : out Boolean)
   --# global out M, Free_Memory, First_ADDR_Available;
   --# derives First_ADDR_Available from Size &
   --#       Free_Memory, M from Size &
   --# Success from Size;
   --# post (Success -> (Size <= DefaultValues.Memory_Size and
   --#       First_ADDR_Available = 0 and
   --#       Free_Memory = Size and
   --#       (M = ←
            HardwareTypes.PhysicalMemory'((HardwareTypes.Memory_Range => ←
            HardwareTypes.Block'(Addr => 0, Sz => Size, St => HardwareTypes.FREE))));)
   is
      begin
         Success := False;
         First_ADDR_Available := 1;
         Free_Memory := 1;
         M := HardwareTypes.PhysicalMemory'((HardwareTypes.Memory_Range => ←
         HardwareTypes.Block'(Addr => 0, Sz => 1, St => HardwareTypes.FREE));
         if (Size >= 1 and Size <= DefaultValues.Memory_Size) then
            First_ADDR_Available := 0;
            Free_Memory := Size;
            M := HardwareTypes.PhysicalMemory'((HardwareTypes.Memory_Range => ←
            HardwareTypes.Block'(Addr => 0, Sz => Size, St => HardwareTypes.FREE));
            Success := True;
         end if;
         end Init;
      end Hardware;

CODE LISTING C.11: Hardware body package

C.9 CMS

Specification

with PartitionTypes, SYT, Hardware, DefaultValues, ConfigValues, SEF, ←
   ErrorTypes, TablesTypes;
use type PartitionTypes.CommunicationMode;
   --# inherit PartitionTypes, SYT, Hardware, DefaultValues, ConfigValues, SEF, ←
      ErrorTypes, TablesTypes;
package CMS is
   procedure InitSystem (Status : out Boolean);
   --# global in out SYT.Communications_Table, ConfigValues.Partition_State, ←
      ConfigValues.FileState, ConfigValues.Size_Of_Blocks_For_Communication, ←
      SEF.Errors_Table;
   --# in out SYT.Partitions_Table;
   --# out Hardware_Mem;
   --# in out SYT.PartitionsExecutionSequence_Table;
   --# in out ConfigValues.Partitions_Execution_Sequence;
Appendix C. SPARK packages

12  --#  in out SYT.PIFF_Table;
13  --#  in out ConfigValues.PIFF_Line;
14  --# derives SYT.Partitions_Table from SYT.Partitions_Table, ←
15                  ConfigValues.FileState &
16  --#  SYT.Communications_Table from SYT.Communications_Table, ←
17                  ConfigValues.Size_Of_Blocks_For_Communication, ConfigValues.FileState &
18  --#  Hardware.Nem from ←
19                  ConfigValues.Size_Of_Blocks_For_Communication, ConfigValues.FileState &
20  --#  ConfigValues.Partition_State from ←
21                  ConfigValues.FileState from ConfigValues.FileState &
22  --#  ConfigValues.Size_Of_Blocks_For_Communication from ←
23                  ConfigValues.FileState from ConfigValues.FileState &
24  --#  SEF.Errors_Table from SEF.Errors_Table, ←
25                  ConfigValues.FileState &
26  --#  Status from ConfigValues.FileState &
27  --#  SYT.PartitionsExecutionSequence_Table from ←
28                  SYT.PartitionsExecutionSequence_Table, ConfigValues.FileState &
29  --#  ConfigValues.Partitions_Execution_Sequence from ←
30                  ConfigValues.FileState, ConfigValues.Partitions_Execution_Sequence &
31  --#  SYT.PIFF_Table from SYT.PIFF_Table, ConfigValues.FileState, ←
32                  ConfigValues.PIFF_Line &
33  --#  ConfigValues.PIFF_Line from ConfigValues.PIFF_Line, ←
34                  ConfigValues.FileState;
35
36  procedure RunProcesses (currentPartition_ID : in ←
37                  PartitionTypes.PartitionsNumber;
38  OK : out Boolean);  
39  --# global in out SYT.Communications_Table, SYT.Partitions_Table, ←
40                  SEF.Errors_Table;
41  --# derives SYT.Communications_Table from SYT.Partitions_Table, ←
42                  SYT.Communications_Table, currentPartition_ID &
43  --#  OK from currentPartition_ID, SYT.Communications_Table, ←
44                  SYT.Partitions_Table &
45  --#  SYT.Partitions_Table from SYT.Partitions_Table, ←
46                  SYT.Communications_Table, currentPartition_ID &
47  --#  SEF.Errors_Table from SEF.Errors_Table, SYT.Partitions_Table, ←
48                  SYT.Communications_Table, currentPartition_ID; 
49
50  procedure ContextSwitch (currentPartition_ID : in ←
51                  PartitionTypes.PartitionsNumber;
52  nextPartition_ID : in ←
53                  PartitionTypes.PartitionsNumber);  
54  --# global in out SYT.Communications_Table;
55  --#  in out SYT.Partitions_Table;
56  --#  in SYT.PIFF_Table;
57  --# derives SYT.Communications_Table from nextPartition_ID, ←
58                  SYT.Communications_Table, currentPartition_ID, SYT.PIFF_Table &
59  --#  SYT.Partitions_Table from SYT.Partitions_Table, ←
60                  nextPartition_ID, SYT.Communications_Table, currentPartition_ID, ←
61                  SYT.PIFF_Table;
62
63 end CMS;

Code Listing C.12: CMS specification package
Body

package body CMS is

procedure InitSystem (Status : out Boolean) is
begin
-# accept Flow_Message, 22, "Invariant";
Hardware.Init (Defaults.Memory_Size, Status);
if Status then
ConfigValues.Open (Status);
if Status then
SYT.InitPartitionsTable;
SYT.InitCommunicationsTable;
SYT.InitPIFPTable;
ConfigValues.Close (Status);
else
SEF.Error (ErrorTypes.Configuration);
end if;
else
SEF.Error (ErrorTypes.Hardware);
end if;
end InitSystem;

procedure ContextSwitch (currentPartition_ID : in ← PartitionTypes.PartitionsNumber;
nextPartition_ID : in ← PartitionTypes.PartitionsNumber) is
begin
SYT.DeleteCommunicationForPartition (currentPartition_ID);
SYT.CheckPIFP (nextPartition_ID);
SYT.UpdateCommunicationList (nextPartition_ID);
SYT.ChangePartitionMode (currentPartition_ID, PartitionTypes.IDLE);
SYT.ChangePartitionMode (nextPartition_ID, PartitionTypes.NORMAL);
end ContextSwitch;

procedure RunProcesses (currentPartition_ID : in ← PartitionTypes.PartitionsNumber;
OK : out Boolean) is
begin
for i in PartitionTypes.Index_Range_Processes_Per_Partition loop
-# assert i in PartitionTypes.Index_Range_Processes_Per_Partition;
case ← SYT.Partitions.Table (SYT.GetPartitionIndex (currentPartition_ID)).Processes (i).operation ← is
when PartitionTypes.OP_Communication ⇒ ←
SYT.Communication (currentPartition_ID, i, OK);
when PartitionTypes.OP_ReadWrite ⇒ ←
SYT.ReadWrite (SYT.GetPartitionIndex (currentPartition_ID), i, OK);
when others ⇒ OK := True;
end case;
end loop;
end RunProcesses;
end CMS;

Code Listing C.13: CMS body package

C.10 SYT

Specification

with TablesTypes, PartitionTypes, ConfigValues, PRT, HardwareTypes, Hardware, SEF, ErrorTypes, DefaultValues;

use type PartitionTypes . CommunicationMode;

---# inherit TablesTypes , PartitionTypes , ConfigValues , PRT , HardwareTypes , Hardware , SEF , ErrorTypes , DefaultValues;

package SYT

---# own Partitions_Table, Communications_Table, PIFP_Table, PartitionsExecutionSequence_Table;

---# initializes Partitions_Table, Communications_Table, PIFP_Table, PartitionsExecutionSequence_Table;

is

Partitions_Table : TablesTypes . PartitionsTable :=
TablesTypes . PartitionsTable ' ( PartitionTypes . PartitionsNumber =>
PartitionTypes . Partition ' ( ID => PartitionTypes . IdType ' First ,
MemoryBounds => PartitionTypes . Tuple ' ( init =>
PartitionTypes . AddrAllowValues ' First , sz =>
PartitionTypes . PartitionSizeAllowValues ' First ) , Duration =>
Positive ' first , Mode => PartitionTypes . Partition_Mode ' First , Processes =>
=>
PartitionTypes . Processes_List ' ( PartitionTypes . Index_Range_Processes_Per_Partititon =>
=> PartitionTypes . ProcessesType ' ( operation =>
PartitionTypes . OperationType ' First , blk =>
PartitionTypes . Tuple2 ' ( init =>
PartitionTypes . AddrAllowValues ' First , sz =>
PartitionTypes . ReadWriteAllowValues ' First ) , mode =>
PartitionTypes . CommunicationMode ' First , size =>
PartitionTypes . SizeOfCommunication ' First , to =>
PartitionTypes . PartitionsNumber ' First ) ) , Com =>
PartitionTypes . Communication ' ( PartitionTypes . Index_Range =>
PartitionTypes . Communication_Mode ' Last ) );

Communications_Table : TablesTypes . CommunicationsTable :=
TablesTypes . CommunicationsTable ' ( TablesTypes . CommunicationsBlocksNumber =>
=> TablesTypes . CommunicationState ' ( Blk =>
TablesTypes . TupleCommunication ' ( init =>
PartitionTypes . AddrAllowValues ' First , sz =>
PartitionTypes . CommunicationsBlockSizeAllowValues ' First ) , State =>
TablesTypes . Blocks_First , From_Partition =>
PartitionTypes . IdType ' First , To_Partition =>
PartitionTypes . PartitionsNumber ' First , Mode =>
PartitionTypes . CommunicationMode ' Last , FreeSize =>
PartitionTypes . CommunicationsBlockSizeAllowValues ' First ) );
PIFP_Table : TablesTypes.PIFPTable := PIFPTable' (TablesTypes.PartitionTypes.PartitionsNumber =>
(TablesTypes.Partitions' (TablesTypes.PartitionsNumber =>
TablesTypes.FlowMode 'Last)));

PartitionsExecutionSequence_Table : TablesTypes.PartitionsExecutionSequenceTable :=
(TablesTypes.PartitionsExecutionIndex =>
TablesTypes.PartitionsExecutionIndex 'First');

procedure InitPartitions_Table;
  --# global out Partitions_Table;
  --# in out ConfigValues.Partition_State;
  --# in out Hardware_Mem;
  --# in out ConfigValues.FileState;
  --# in out ConfigValues.Size_Of_Black_For_Comunication;
  --# in out SEF.Errors_Table;
  --# in out PartitionsExecutionSequence_Table;
  --# in ConfigValues.Partitions.Execution_Sequence;
  --# derives Partitions_Table from Hardware_Mem, ConfigValues.FileState &
                Hardware_Mem from Hardware_Mem, ConfigValues.FileState &
                ConfigValues.Partition_State from ConfigValues.FileState &
                ConfigValues.FileState from ConfigValues.FileState &
                SEF.Errors_Table from SEF.Errors_Table, ConfigValues.FileState &
                PartitionsExecutionSequence_Table from ConfigValues.FileState &
                ConfigValues.Partitions.Execution_Sequence from ConfigValues.FileState &
                ConfigValues.Size_Of_Black_For_Comunication from ConfigValues.FileState &
procedure InitCommunications_Table;
  --# global in out Communications_Table;
  --# in out Hardware_Mem;
  --# in ConfigValues.Size_Of_Black_For_Comunication;
  --# derives Communications_Table from Communications_Table, Hardware_Mem, ConfigValues.Size_Of_Black_For_Comunication &
                Hardware_Mem from Hardware_Mem, ConfigValues.Size_Of_Black_For_Comunication;
procedure InitPIFP_Table;
  --# global in out PIFP_Table;
  --# in out ConfigValues.FileState;
  --# in out ConfigValues.PIFP_Line;
  --# in out SEF.Errors_Table;
  --# derives configValues.PIFP_Line from configValues.PIFP_Line, ConfigValues.FileState &
                PIFP_Table from PIFP_Table, configValues.PIFP_Line, ConfigValues.FileState &
                ConfigValues.FileState from ConfigValues.FileState &
                SEF.Errors_Table from SEF.Errors_Table, ConfigValues.FileState;
Appendix C. SPARK packages

function GetPartitionIndex (ID : PartitionTypes.PartitionsNumber) return PartitionTypes.PartitionsNumber;
--# global Partitions_Table;
--# return M => (Partitions_Table(M).ID = ID -> M = M);

procedure UpdateCommunicationList (P_ID : in PartitionTypes.PartitionsNumber);  
--# global in Communications_Table;
--# in out Partitions_Table;
--# derives Partitions_Table from Partitions_Table, P_ID, -> Communications_Table;

procedure Communication (PIDFrom : in PartitionTypes.PartitionsNumber; 
--# global in out Communications_Table;
--# in out Partitions_Table;
--# derives Communications_Table from Partitions_Table, Communications_Table, PIDFrom, Index & Communications_Table, PIDFrom, Index &
--# Partitions_Table from Partitions_Table, Communications_Table, Communications_Table, PIDFrom, Index &
--# SEF.Errors_Table from SEF.Errors_Table, Partitions_Table, Communications_Table, PIDFrom, Index;

procedure DeleteCommunicationForPartition (ID_Partition : in PartitionTypes.IdType);  
--# global in out Communications_Table;
--# derives Communications_Table from Communications_Table, ID_Partition;

procedure ReadWrite (PID : in PartitionTypes.PartitionsNumber; Index : in PartitionTypes.Index.Range.Processes_Per_Partition; OK : out Boolean);
--# global in out Partitions_Table;
--# in out SEF.Errors_Table;
--# derives OK from PID, Index, Partitions_Table &
--# Partitions_Table from Partitions_Table, PID, Index &
--# SEF.Errors_Table from SEF.Errors_Table, Partitions_Table, ->

procedure CheckPIFP(ID_Partiton : in PartitionTypes.IdType);  
--# global in out Communications_Table;
--# in PIFP_Table;
--# derives Communications_Table from Communications_Table, ID_Partiton, -> PIFP_Table;

procedure ChangePartitionMode(ID_Partition : in PartitionTypes.PartitionsNumber; Mode : in PartitionTypes.Partition_Mode);
--# global in out Partitions_Table;
--# derives Partitions_Table from Partitions_Table, ID_Partition, Mode;

function GetDurationPRT(ID_Partition : PartitionTypes.PartitionsNumber) return PartitionTypes.Partition_Duration;
--# global Partitions_Table;
--# return Partitions_Table(GetPartitionIndex(ID_Partition, <-
Partitions_Table)).Duration;
end SYT;

CODE LISTING C.14: SYT specification package

Body

package body SYT is

procedure InitPartitionsTable is
  Partition : PartitionTypes.Partition;
  Status : Boolean;
  OK : Boolean;
  ADDR : HardwareTypes.MemoryRange;
begin
  Partitions,Table :=
    TablesTypes.PartitionsTable'(PartitionTypes.PartitionsNumber => <-
    PartitionTypes.Partition'
    (ID => PartitionTypes.IdType'First,
     MemoryBounds => <-
     PartitionTypes.Tuple'(init => PartitionTypes.AddrAllowValues'First,sz =>
     => PartitionTypes.PartitionSizeAllowValues'First),
     Duration => Positive 'first,
     Mode => <-
     PartitionTypes.Partition_Mode'First,
     Processes => <-
     PartitionTypes.Processes_List'(
     PartitionTypes.Index_Range_Processes_Per_Partition <=
     => PartitionTypes.ProcessesType'
     (operation => PartitionTypes.OperationType'First,
      blk => PartitionTypes.Tuple2'(init => <-
      PartitionTypes.AddrAllowValues'First,sz => <-
      PartitionTypes.ReadWriteAllowValues'First),
      mode => PartitionTypes.CommunicationMode'First,
      size => PartitionTypes.SizeOfCommunication'First,
      to => PartitionTypes.PartitionsNumber 'First)),
    Com => <-
    PartitionTypes.Communication'(PartitionTypes.Index_Range => <-
    PartitionTypes.Communication_Mode'Last) ));
  PartitionsExecutionSequence_Table := <-
  TablesTypes.PartitionsExecutionSequenceTable'(TablesTypes.PartitionsExecutionIndex <=
  => TablesTypes.PartitionsExecutionIndex 'First);
  ConfigValues.Read_Size_Of_Blocks_For_Communication(Status);
end InitPartitionsTable;
if status then
    for i in PartitionTypes.PartitionsNumber loop
        --# assert i <= PartitionTypes.PartitionsNumber'last and i >= PartitionTypes.PartitionsNumber'first;
        PartitionsTable(i) := Partition;
        ConfigValues.Read_PRT_Values(Status);
        if Status then
            Hardware.AllocBlock(ConfigValues.Get_Partition_Size, ADDR, OK);
            if OK then
                PartitionsTable(i) := Partition;
            else
                SEF.Error(ErrorTypes.Hardware);
            end if;
        else
            SEF.Error(ErrorTypes.Configuration);
        end if;
    end loop;
    ConfigValues.Read_PRT_Exec_Sequence(Status);
    if Status then
        PartitionsExecutionSequence_Table := ConfigValues.Partitions_Execution_Sequence;
    else
        SEF.Error(ErrorTypes.Configuration);
    end if;
else
    SEF.Error(ErrorTypes.Configuration);
end if;
end InitPartitionsTable;

procedure InitCommunicationsTable is
    OK : Boolean;
    ADDR : HardwareTypes.Memory_Range;
    begin
        for i in TablesTypes.CommunicationsBlocksNumber loop
            --# assert i <= TablesTypes.CommunicationsBlocksNumber'last and i >= TablesTypes.CommunicationsBlocksNumber'first;
            Hardware.AllocBlock(ConfigValues.Size_Of_Blocks_For_Communication, ADDR, OK);
            if OK then
                Communications_Table(i).Blk := Table_Communication'(init => ADDR, sz => ConfigValues.Size_Of_Blocks_For_Communication);
                Communications_Table(i).State := TablesTypes.FREE;
                Communications_Table(i).From_Partition := PartitionTypes.IdType'First;
                Communications_Table(i).To_Partition := PartitionTypes.PartitionsNumber'First;
                Communications_Table(i).Mode := PartitionTypes.READ;
                Communications_Table(i).FreeSize := Communications_Table(i).Blk.sz;
            end if;
        end loop;
    end InitCommunicationsTable;
Appendix C. SPARK packages

---

```plaintext
end loop;
end InitCommunicationsTable;

procedure InitPifPTable is
  Status : Boolean;
  ID : PartitionTypes.PartitionsNumber;
begin
  ConfigValues.Read_PiP(Status);
  if Status then
    for i in PartitionTypes.PartitionsNumber loop
      ConfigValues.Read_PiP_Line(Status, ID);
      if Status then
        PiP_Table(ID) := ConfigValues.PiP_Line;
      else
        SEF.Error(ErrorTypes.Application);
      end if;
    end loop;
  else
    SEF.Error(ErrorTypes.Configuration);
  end if;
end InitPifPTable;

function GetPartitionIndex (ID : PartitionTypes.PartitionsNumber) return PartitionTypes.PartitionsNumber is
  Index : PartitionTypes.PartitionsNumber;
begin
  Index := PartitionTypes.PartitionsNumber 'First';
  for i in PartitionTypes.PartitionsNumber loop
    if PiP_Table(i).ID = ID then
      Index := i;
    end if;
  end loop;
  return Index;
end GetPartitionIndex;

procedure UpdateCommunicationList (P_ID : in PartitionTypes.PartitionsNumber)
  is
    function CommunicationsTableStateIsFree(Index : in TableTypesCommunicationsBlocksNumber) return Boolean
      --# global Communications_Table;
    is
      r : Boolean;
    begin
      case Communications_Table(Index).State is
        When TableTypes.FREE => r := True;
        When TableTypes.OCCUPIED => r := False;
      end case;
      return r;
    end CommunicationsTableStateIsFree;
```
function Get_PartitionIndex (ID : PartitionTypes.PartitionsNumber) ↵
return PartitionTypes.PartitionsNumber
--# global Partitions_Table;
is
  Index : PartitionTypes.PartitionsNumber;
begin
  Index := PartitionTypes.PartitionsNumber 'First;
  for i in PartitionTypes.PartitionsNumber loop
    --# assert i in PartitionTypes.PartitionsNumber;
    if Partitions_Table(i).ID = ID then
      Index := i;
      exit;
    end if;
  end loop;
  return Index;
end Get_PartitionIndex;

begin
  for i in PartitionTypes.Index_Range loop
    --# assert i <= PartitionTypes.Index_Range 'last and i >= 1;
    if CommunicationsTableStateIsFree(i) then
      Partitions_Table(Get_PartitionIndex(P_ID)).Com(i) := PartitionTypes.FREE;
    else
      if (not CommunicationsTableStateIsFree(i)) and
         (Communications_Table(i).To_Partition = P_ID) then
        if Communications_Table(i).Mode = PartitionTypes.WRITE then
          Partitions_Table(Get_PartitionIndex(P_ID)).Com(i) := PartitionTypes.WRITE;
        else
          Partitions_Table(Get_PartitionIndex(P_ID)).Com(i) := PartitionTypes.BLOCKED;
        end if;
      else
        Partitions_Table(Get_PartitionIndex(P_ID)).Com(i) := PartitionTypes.READ;
      end if;
    end if;
  end loop;
end UpdateCommunicationList;

procedure Communication (PIDFrom : in PartitionTypes.PartitionsNumber; ↵
Index : in PartitionTypes.Index_Range_Processes_Per_Partition ; OK : out Boolean) is
  CanSend : Boolean;
tempTo : PartitionTypes.PartitionsNumber;
tempMode : PartitionTypes.CommunicationMode;
tempSize : PartitionTypes.SizeOfCommunication2;
begin
  CanSend := true;
  tempTo := PIDFrom;
  tempMode := PartitionTypes.READ;
  tempSize := PartitionTypes.SizeOfCommunication2;
  procedure UpdateCommunicationsTable(Index : in PartitionTypes.PartitionsNumber; ↵
ID_PTo : in PartitionTypes.PartitionsNumber;
M : in \leftarrow
\text{PartitionTypes. CommunicationMode;}
S : in \leftarrow
\text{PartitionTypes. SizeOfCommunication;}
#.global in out Communications_Table;
# derives Communications_Table from Communications_Table, Index, \leftarrow PIDFrom, ID_PTo, M, S;
is
begin
Communications_Table(Index).Blk := Communications_Table(Index).Blk;
Communications_Table(Index).State := TablesTypes.OCCUPIED;
Communications_Table(Index).From_Partition := PIDFrom;
Communications_Table(Index).To_Partition := ID_PTo;
Communications_Table(Index).Mode := M;
if (((Communications_Table(Index).FreeSize) - S) >= \leftarrow DefaultValues.CommunicationsBlockSizeMinValue) and \leftarrow ((Communications_Table(Index).FreeSize) - S) <= \leftarrow DefaultValues.CommunicationsBlockSizeMaxValue) then
Communications_Table(Index).FreeSize := \leftarrow Communications_Table(Index).FreeSize - S;
end if;
end UpdateCommunicationsTable;

procedure CanSendMessage(Index : out PartitionTypes.IndexRange; OK : out Boolean;
ID_PTo : in PartitionTypes.PartitionsNumber;
M : in PartitionTypes.CommunicationMode;
S : in PartitionTypes.SizeOfCommunication;)
# global in PIDFrom, Communications_Table, Partitions_Table;
# derives Index from PIDFrom, M, S, Communications_Table, ID_PTo, \leftarrow Partitions_Table &
# OK from PIDFrom, M, S, Communications_Table, ID_PTo, \leftarrow Partitions_Table;
is
function WriteBlockHaveSpace(Index : \leftarrow TablesTypes.CommunicationsBlocksNumber) return Boolean
# global Partitions_Table, PIDFrom, M, S, Communications_Table, \leftarrow ID_PTo;
is
r : Boolean;
begin
case Partitions_Table(GetPartitionIndex(PIDFrom)).Com(Index) is
when PartitionTypes.WRITE \Rightarrow
if Communications_Table(Index).Mode = M and \leftarrow (Communications_Table(Index).FreeSize >= S) and \leftarrow (Communications_Table(Index).To_Partition = ID_PTo)
then r := True;
else r := False;
end if;
when others \Rightarrow r := False;
end case;
return r;
end WriteBlockHaveSpace;
function BlockIsFree(\text{Index} : \leftarrow \text{TablesTypes}.\text{CommunicationsBlocksNumber}) \ return \ Boolean
\begin{verbatim}
begin
case \text{Partitions_Table}(\text{GetPartitionIndex(PIDFrom)})\text{.Com(\text{Index})} \ is
  when \text{PartitionTypes.FREE} => r := True;
  when others => r := False;
end case;
\end{verbatim}
\end{verbatim}
\end{verbatim}
\begin{verbatim}
return r;
end BlockIsFree;
\end{verbatim}

begin
\begin{verbatim}
OK := False;
Index := 1;
for i in \text{PartitionTypes.Index\_Range} loop
  --\# assert 1 <= \text{PartitionTypes.Index\_Range\_last} and i >= 1;
  if \text{WriteBlockHaveSpace(i)} then
    OK := True;
    Index := i;
    exit;
  end if;
  if \text{BlockIsFree(i)} then
    OK := True;
    Index := i;
    exit;
  end if;
end loop;
end CanSendMessage;
\end{verbatim}

procedure SendMessageSplit(OK : \ out \ Boolean;
\begin{verbatim}
ID_PTo : \ in \ \text{PartitionTypes}.\text{PartitionsNumber};
M : \ in \ \text{PartitionTypes}.\text{CommunicationMode};
S : \ in \ \text{PartitionTypes}.\text{SizeOfCommunication})
\begin{verbatim}
begin
  --\# global in out \text{Communications\_Table}, \text{Partitions\_Table};
  --\# in \text{PIDFrom};
  --\# derives OK from \text{PIDFrom}, M, S, \text{Communications\_Table}, \leftarrow \text{ID_PTo}, \text{Partitions\_Table} &
  --\# \text{Communications\_Table} from M, S, \text{ID_PTo}, \text{Partitions\_Table}, \leftarrow \text{Communications\_Table}, \text{PIDFrom} &
  --\# \text{Partitions\_Table} from M, S, \text{Communications\_Table}, \text{ID_PTo}, \leftarrow \text{PIDFrom}, \text{Partitions\_Table};
  is
  SizeLeft, SizeLeft\_Temp : \text{PartitionTypes}.\text{SizeOfCommunication2};
  \text{Partitions\_Table\_Temp} : \text{TablesTypes}.\text{PartitionsTable};
  \text{Communications\_Table\_Temp} : \text{TablesTypes}.\text{CommunicationsTable};
  Flag : \ Boolean;
\end{verbatim}
\begin{verbatim}
  procedure WriteBlockHaveSpace(\text{Index} : \ in \leftarrow \text{TablesTypes}.\text{CommunicationsBlocksNumber}; \ r : \ out \ Boolean)
  --\# global in out \text{Communications\_Table\_Temp};
Appendix C. SPARK packages

```plaintext
254 --# in PIDFrom, M, ID_PTo, Partitions_Table, SizeLeft_Temp, ←
255 Partitions_Table_Temp;
256 --# derives r from Index, PIDFrom, M, ID_PTo, Partitions_Table, ←
257 SizeLeft_Temp, Partitions_Table_Temp, Communications_Table_Temp &
258 --# Communications_Table_Temp from Partitions_Table_Temp, ←
259 Partitions_Table, Communications_Table_Temp, PIDFrom, M, ID_PTo, ←
260 SizeLeft_Temp, Index;

261 begin
262 case ←
263 Partitions_Table_Temp( GetPartitionIndex(PIDFrom) ).Com(Index) is
264 when PartitionTypes.WRITE =>
265 if Communications_Table_Temp(Index).Mode = M and
266 ( Communications_Table_Temp(Index).FreeSize >= ←
267 SizeLeft_Temp ) and
268 ( Communications_Table_Temp(Index).To_Parition = ID_PTo )
269 then r := True;
270 else
271 if Communications_Table_Temp(Index).Mode = PartitionTypes.READ and
272 ( Communications_Table_Temp(Index).FreeSize >= ←
273 SizeLeft_Temp ) and
274 ( Communications_Table_Temp(Index).From_Parition = ←
275 PIDFrom )
276 then
277 Communications_Table_Temp(Index).From_Parition := ID_PTo;
278 Communications_Table_Temp(Index).To_Parition := PIDFrom;
279 r := True;
280 end if;
281 when others => r := False;
282 end if;
283 end case;
284 end WriteBlockHaveSpace;

285 function BlockIsFree(Index : ←
286 TablesTypes.Com municationsBlocksNumber) return Boolean
287 --# global PIDFrom, Partitions_Table_Temp, Partitions_Table;
288 is
289 r : Boolean;
290 begin
291 case ←
292 Partitions_Table_Temp( GetPartitionIndex(PIDFrom) ).Com(Index) is
293 when PartitionTypes.FREE => r := True;
294 when Others => r := False;
295 end case;
296 return r;
297 end BlockIsFree;

298 procedure UpdateFreeSize
299 --# global in out Communications_Table_Temp;
```
```spArk
begin
for i in tablesTypes.CommunicationsBlocksNumber loop
  if Communications_Table.Temp(i).Mode = PartitionTypes.READ then
    Communications_Table.Temp(i).FreeSize := <-
    Communications_Table.Temp(i).Blk.sz;
  end if;
end loop;
end

begin
SizeLeft := S;
SizeLeft_Temp := S;
Partitions_Table_Temp := Partitions_Table;
Communications_Table_Temp := Communications_Table;
loop
  if Communications_Table_Temp(1).Blk.sz >= SizeLeft_Temp then
    Flag := False;
    for i in PartitionTypes.Index.Range loop
      if (Communications_Table_Temp(i).State = TablesTypes.OCCUPIED) and
         Communications_Table_Temp(i).Mode = PartitionTypes.WRITE then
        if (SizeLeft_Temp - SizeLeft_Temp) >= 0 then
          SizeLeft := SizeLeft_Temp - SizeLeft_Temp;
          SizeLeft_Temp := SizeLeft;
        end if;
      else
        if BlockIsFree(i) then
          Flag := True;
          OK := True;
          Communications_Table_Temp(i).Blk := <-
          Communications_Table_Temp(i).State := <-
          TablesTypes.OCCUPIED;
        end if;
      end if;
    end loop;
  end if;
end loop;
end
```

---

```
--# derives Communications_Table_Temp from Communications_Table_Temp;

is
begin
  for i in tablesTypes.CommunicationsBlocksNumber loop
    if Communications_Table_Temp(i).Mode = PartitionTypes.READ then
      Communications_Table_Temp(i).FreeSize := <-
      Communications_Table_Temp(i).Blk.sz;
    end if;
  end loop;
end

begin
SizeLeft := S;
SizeLeft_Temp := S;
Partitions_Table_Temp := Partitions_Table;
Communications_Table_Temp := Communications_Table;
loop
  if Communications_Table_Temp(1).Blk.sz >= SizeLeft_Temp then
    Flag := False;
    for i in PartitionTypes.Index.Range loop
      if (Communications_Table_Temp(i).State = TablesTypes.OCCUPIED) and
         Communications_Table_Temp(i).Mode = PartitionTypes.WRITE then
        if (SizeLeft_Temp - SizeLeft_Temp) >= 0 then
          SizeLeft := SizeLeft_Temp - SizeLeft_Temp;
          SizeLeft_Temp := SizeLeft;
        end if;
      else
        if BlockIsFree(i) then
          Flag := True;
          OK := True;
          Communications_Table_Temp(i).Blk := <-
          Communications_Table_Temp(i).State := <-
          TablesTypes.OCCUPIED;
        end if;
      end if;
    end loop;
  end if;
end loop;
end
```

---


\[\text{Communications.Table.Temp(i).From Partition := PIDFrom;}\]
\[\text{Communications.Table.Temp(i).To Partition := ID.PTo;}\]
\[\text{Communications.Table.Temp(i).Mode := M;}\]
\[\text{if Communications.Table.Temp(i).FreeSize >= SizeLeft.Temp then}\]
\[\text{Communications.Table.Temp(i).FreeSize := SizeLeft.Temp;}\]
\[\text{end if;}\]
\[\text{Partitions.Table.Temp(GetPartitionIndex(PIDFrom)).Com(i) := PartitionTypes.WRITE;}\]
\[\text{if (SizeLeft - SizeLeft.Temp) >= 0 then}\]
\[\text{SizeLeft := SizeLeft - SizeLeft.Temp;}\]
\[\text{SizeLeft.Temp := SizeLeft;}\]
\[\text{end if;}\]
\[\text{end if;}\]
\[\text{end loop;}\]
\[\text{else}\]
\[\text{SizeLeft.Temp := SizeLeft.Temp - 1;}\]
\[\text{OK := True;}\]
\[\text{end if;}\]
\[\text{if SizeLeft = 0 then}\]
\[\text{UpdateFreeSize;}\]
\[\text{Partitions.Table := Partitions.Table.Temp;}\]
\[\text{Communications.Table := Communications.Table.Temp;}\]
\[\text{end if;}\]
\[\text{exit when SizeLeft.Temp = 0 or (not OK);}\]
\[\text{end loop;}\]
\[\text{end SendMessageSplit;}\]

\begin{verbatim}
begin
  tempTo := Partitions.Table(GetPartitionIndex(PIDFrom)).Processes(Index).to;
  tempMode := Partitions.Table(GetPartitionIndex(PIDFrom)).Processes(Index).mode;
  tempSize := Partitions.Table(GetPartitionIndex(PIDFrom)).Processes(Index).size;
  SendMessageSplit(CanSend, tempTo, tempMode, tempSize);
  if CanSend then
    OK := True;
  else
    SEF.Error(ErrorTypes.Application);
    Partitions.Table(GetPartitionIndex(PIDFrom)).Mode := PartitionTypes.ERROR;
  end if;
end Communication;
\end{verbatim}
Appendix C. **SPARK packages**

```plaintext
procedure DeleteCommunicationForPartition (ID_Partiton : in PartitionTypes.IdType) is
begin
  for i in Tablestypes.CommunicationsBlocksNumber loop
    --# assert i in Tablestypes.CommunicationsBlocksNumber;
    if Communications_Table(i).To_Partition = ID_Partiton then
      Communications_Table(i).State := Tablestypes.FREE;
      Communications_Table(i).From_Partition := PartitionTypes.IdType 'First;
      Communications_Table(i).To_Partition := PartitionTypes.PartitionsNumber 'First;
      Communications_Table(i).Mode := PartitionTypes.READ;
      Communications_Table(i).FreeSize := Communications_Table(i).Blk.sz;
    end if;
  end loop;
end DeleteCommunicationForPartition;

procedure ReadWrite (PID: in PartitionTypes.PartitionsNumber; Index: in PartitionTypes.Index_Range_Processes_Per_Partition; OK: out Boolean) is
begin
  OK := False;
  if (Partitions_Table(PID).MemoryBounds.init <= PartitionTypes.Table(Index).blk.init and
  >= (Partitions_Table(PID).MemoryBounds.init + Partitions_Table(PID).MemoryBounds.sz) <=
  (Partitions_Table(PID).MemoryBounds.init + PartitionTypes.Table(Index).blk.init))
  then OK := True;
  else
    Partitions_Table(PID).Mode := PartitionTypes.ERROR;
    SEF.Error(ErrorTypes.Application);
  end if;
end ReadWrite;

procedure CheckPifP (ID_Partiton : in PartitionTypes.IdType) is
begin
  for i in Tablestypes.CommunicationsBlocksNumber loop
    --# assert i in Tablestypes.CommunicationsBlocksNumber;
    if (Communications_Table(i).To_Partition = ID_Partiton and
    Communications_Table(i).From_Partition > 0 and
    Communications_Table(i).To_Partition > 0) then
      case (PifP_Table(Communications_Table(i).From_Partition) (Communications_Table(i).To_Partition)) is
        When Tablestypes.N => Communications_Table(i).State := Tablestypes.FREE;
        Communications_Table(i).From_Partition := PartitionTypes.IdType 'First;
        Communications_Table(i).To_Partition := PartitionTypes.PartitionsNumber 'First;
        Communications_Table(i).Mode := PartitionTypes.READ;
        Communications_Table(i).FreeSize := Communications_Table(i).Blk.sz;
    end case;
  end loop;
end CheckPifP;
```
appendix c.

Spark packages

156

when tables types . r => if communications _ table ( i ) . mode /= partition types . read then communications _ table ( i ) . state := tablestypes . free;

partition types . idtype ’ first;

communications _ table ( i ) . to _ partition := partition types . partitions number’ first;

communications _ table ( i ) . mode := partition types . read;

communications _ table ( i ) . freesize := communications _ table ( i ) . blk . sz;

end if;

when tables types . w => if communications _ table ( i ) . mode /= partition types . write then communications _ table ( i ) . state := tablestypes . free;

partition types . idtype ’ first;

communications _ table ( i ) . to _ partition := partition types . partitions number’ first;

communications _ table ( i ) . mode := partition types . read;

communications _ table ( i ) . freesize := communications _ table ( i ) . blk . sz;

end if;

when tables types . rw => null;

end case;

end if;

end loop;

end check pifp;

procedure changepartitionmode ( id _ partition : in <-

partition types . partitions number; mode : in <-

partition types . partition _ mode ) is

begin

<- p rt . setpartitionmode ( partitions _ table ( getpartitionindex ( id _ partition ) ), <-

mode);

end changepartitionmode;

function getdurationprt ( id _ partition : partition types . partitions number) <-

return partition types . partition _ duration is

begin

return partitions _ table ( getpartitionindex ( id _ partition ) ) . duration;

end getdurationprt;

end syt;

code listing c.15: syt body package

c.11 configvalues

specification

with partition types , tables types ;
use type partition types . operationtype;
--# inherit partition types , tables types , file , defaultvalues;
package ConfigValues

--# own Partition_State, Size_Of_Blocks_For_Communication, PIFP_Line, ←
--# initializes FileState, Size_Of_Blocks_For_Communication, ←
--# Partitions_Execution_Sequence, Partition_State, PIFP_Line;

is

Size_Of_Blocks_For_Communication : ←
PartitionTypes . CommunicationsBlockSizeAllowValues := ←
PartitionTypes . CommunicationsBlockSizeAllowValues ' First ;
PIFP_Line : TablesTypes . Partitions := ←
TablesTypes . Partitions '{ PartitionTypes . PartitionsNumber ⇒ ←
TablesTypes . FlowMode ' Last ) ;
Partitions_Execution_Sequence : ←
TablesTypes . PartitionsExecutionSequenceTable :=
←
TablesTypes . PartitionsExecutionSequenceTable '{ TablesTypes . PartitionsExecutionIndex ←
⇒ TablesTypes . PartitionsExecutionIndex ' First ) ;

procedure ValidateFile ( Success : out Boolean ) ;
--# global in out FileState;
--# derives Success, FileState from FileState;

procedure Open ( OK : out Boolean ) ;
--# global in out FileState;
--# derives FileState, OK from FileState;

procedure Close ( OK : out Boolean ) ;
--# global in out FileState;
--# derives FileState, OK from FileState;

procedure Read_PRT_Values ( Success : out Boolean ) ;
--# global out Partition_State;
--# in out FileState;
--# derives Success, FileState from FileState &
--# Partition_State from FileState;

function Get_Partition_ID return PartitionTypes . IdType ;
--# global Partition_State;

function Get_Partition_Size return PartitionTypes . PartitionSizeAllowValues ;
--# global Partition_State;

function Get_Partition_Duration return PartitionTypes . Partition_Duration ;
--# global Partition_State;

function Get_Partition_Mode return PartitionTypes . Partition_Mode ;
--# global Partition_State;

function Get_Partition_Processes return PartitionTypes . Processes_List ;
--# global Partition_State;
function Get_Size_Of_Blocks_For_Communication return ←
  PartitionTypes.CommunicationsBlockSizeAllowValues;
  ← PartitionTypes.CommunicationsBlockSizeAllowValues;
procedure Read_Size_Of_Blocks_For_Communication (Success : out Boolean);
  ← PartitionTypes.CommunicationsBlockSizeAllowValues;
  ← PartitionTypes.CommunicationsBlockSizeAllowValues;
  ← PartitionTypes.CommunicationsBlockSizeAllowValues;
  ← PartitionTypes.CommunicationsBlockSizeAllowValues;
  ← PartitionTypes.CommunicationsBlockSizeAllowValues;
  ← PartitionTypes.CommunicationsBlockSizeAllowValues;
procedure Read_PIPF (Success : out Boolean);
  ← PartitionTypes.PIPFNumber;
  ← PartitionTypes.PIPFNumber;
  ← PartitionTypes.PIPFNumber;
  ← PartitionTypes.PIPFNumber;
  ← PartitionTypes.PIPFNumber;
  ← PartitionTypes.PIPFNumber;
procedure Read_PIPF_Line (Success : out Boolean; ID : out ←
  PartitionTypes.PartitionsNumber);
  ← PartitionTypes.PIPFNumber;
  ← PartitionTypes.PIPFNumber;
  ← PartitionTypes.PIPFNumber;
  ← PartitionTypes.PIPFNumber;
  ← PartitionTypes.PIPFNumber;
procedure Read_PRT_Exec_Sequence (Success : out Boolean);
  ← PartitionTypes.PartitionsExecutionSequence;
  ← PartitionTypes.PartitionsExecutionSequence;
  ← PartitionTypes.PartitionsExecutionSequence;
  ← PartitionTypes.PartitionsExecutionSequence;
  ← PartitionTypes.PartitionsExecutionSequence;
end ConfigValues;

CODE LISTING C.16: ConfigValues specification package

Body

with File, DefaultValues;

package body ConfigValues
  ← PartitionTypes.IdType := PartitionTypes.IdType 'First;
  ← PartitionTypes.PartitionSizeAllowValues 'First;
  ← PartitionTypes.PartitionDuration 'First;
  ← PartitionTypes.PartitionMode 'First;
  ← PartitionTypesProcesses.List :=
  with File, DefaultValues;

package body ConfigValues
  ← PartitionTypes.IdType := PartitionTypes.IdType 'First;
  ← PartitionTypes.PartitionSizeAllowValues 'First;
  ← PartitionTypes.PartitionDuration 'First;
  ← PartitionTypes.PartitionMode 'First;
  ← PartitionTypesProcesses.List :=
Appendix C. SPARK packages

```
  PartitionTypes . Processes _ List '( PartitionTypes . Index _ Range _ Processes _ Per _ Partition on )

  ( PartitionTypes . ProcessesType !=

    ( operation => PartitionTypes . OperationType 'First ,

      blk => PartitionTypes . Tuple2 '( init =>

        PartitionTypes . Addr _ Allow _ Values 'First , sz =>

        PartitionTypes . Read _ Write _ Allow _ Values 'First ) ,

      mode => PartitionTypes . CommunicationMode 'First ,

      size => PartitionTypes . Size _ Of _ Communication 'First ,

      to => PartitionTypes . Partitions _ Number 'First ) )

  subtype Partitions _ Number _ Text _ I is Positive range 1..17;
  subtype Partitions _ Number _ Text _ T is String ( Partitions _ Number _ Text _ I ) ;
  Partitions _ Number _ Title : constant Partitions _ Number _ Text _ T := "Partitions _ Number " ;

  subtype Communication _ Blocks _ Size _ I is Positive range 1..24;
  subtype Communication _ Blocks _ Size _ T is String ( Communication _ Blocks _ Size _ I ) ;
  Communication _ Blocks _ Size _ Title : constant Communication _ Blocks _ Size _ T := "Communication _ Blocks _ Size " ;

  subtype Partition _ I is Positive range 1..9;
  subtype Partition _ T is String ( Partition _ I ) ;
  Partition _ Title : constant Partition _ T := "PARTITION" ;

  subtype Partition _ ID _ I is Positive range 1..12;
  subtype Partition _ ID _ T is String ( Partition _ ID _ I ) ;
  Partition _ ID _ Title : constant Partition _ ID _ T := "Partition _ ID " ;

  subtype Partition _ SIZE _ I is Positive range 1..14;
  subtype Partition _ SIZE _ T is String ( Partition _ SIZE _ I ) ;
  Partition _ SIZE _ Title : constant Partition _ SIZE _ T := "Partition _ SIZE " ;

  subtype Partition _ DURATION _ I is Positive range 1..18;
  subtype Partition _ DURATION _ T is String ( Partition _ DURATION _ I ) ;
  Partition _ DURATION _ Title : constant Partition _ DURATION _ T := "Partition _ DURATION " ;

  subtype Process _ I is Positive range 1..7;
  subtype Process _ T is String ( Process _ I ) ;
  Process _ Title : constant Process _ T := "PROCESS" ;

  subtype Process _ Operation _ I is Positive range 1..17;
  subtype Process _ Operation _ T is String ( Process _ Operation _ I ) ;
  Process _ Operation _ Title : constant Process _ Operation _ T := "Process _ Operation " ;

  subtype Operation _ Text _ I is Positive range 1..13;
  subtype Operation _ Text _ T is String ( Operation _ Text _ I ) ;
```
type OperationStringT is record
  Text : OperationTextT;
  Length : OperationTextI;
end record;

type OperationStringLookUpT is array (PartitionTypes.OperationType) of OperationStringT;
OperationStringLookUp : constant OperationStringLookUpT :=
  (PartitionTypes.OP_Nothing => OperationStringT("Nothing ", 7),
   PartitionTypes.OP_Communication => OperationStringT("Communication", 13),
   PartitionTypes.OP_ReadWrite => OperationStringT("ReadWrite ", 9));

subtype CommunicationModeI is Positive range 1..18;
subtype CommunicationModeT is String (CommunicationModeI);
CommunicationModeTitle : constant CommunicationModeT := "Communication Mode ";

subtype CommunicationModeTextI is Positive range 1..5;
subtype CommunicationModeTextT is String (CommunicationModeTextI);

type CommunicationModeStringT is record
  Text : CommunicationModeTextT;
  Length : CommunicationModeTextI;
end record;

type CommunicationModeStringLookUpT is array CommunicationModeStringT of CommunicationModeStringLookUpT;
CommunicationModeStringLookUp : constant CommunicationModeStringLookUpT :=
  (PartitionTypes.READ => CommunicationModeStringT("Read ", 4),
   PartitionTypes.WRITE => CommunicationModeStringT("Write", 5));

subtype CommunicationSizeI is Positive range 1..18;
subtype CommunicationSizeT is String (CommunicationSizeI);
CommunicationSizeTitle : constant CommunicationSizeT := "Communication Size ";

subtype CommunicationToI is Positive range 1..16;
subtype CommunicationToT is String (CommunicationToI);
CommunicationToTitle : constant CommunicationToT := "CommunicationTo ";

subtype PartitionsExecutionNumberI is Positive range 1..26;
subtype PartitionsExecutionNumberT is String (PartitionsExecutionNumberI);
PartitionsExecutionNumberTitle : constant PartitionsExecutionNumberT := "PartitionsExecutionNumber ";

subtype PartitionsExecutionSequenceI is Positive range 1..27;
subtype PartitionsExecutionSequenceT is String (PartitionsExecutionSequenceI);
Appendix C. **SPARK packages**

```plaintext
PartitionsExecutionSequenceTitle : constant PartitionsExecutionSequenceT := "PartitionsExecutionSequence";

subtype PifpTextI is Positive range 1..4;
subtype PifpTextT is String(PifpTextI);
PifpTitle : constant PifpTextT := "PIFP";

subtype IDTextI is Positive range 1..3;
subtype IDTextT is String(IDTextI);
IDTitle : constant IDTextT := "ID ";

subtype FlowTextI is Positive range 1..3;
subtype FlowTextT is String(FlowTextI);

type FlowStringT is record
  Text : FlowTextT;
  Length : FlowTextI;
end record;

type FlowStringLookUpT is array (TablesTypes.FlowMode) of FlowStringT;
FlowStringLookUp : constant FlowStringLookUpT :=
  FlowStringLookUpT' (TablesTypes.R = FlowStringT'(" R ", 2),
  TablesTypes.W = FlowStringT'(" W ", 2),
  TablesTypes.RW = FlowStringT'(" RW", 3),
  TablesTypes.N = FlowStringT'(" N ", 2));

ConfigFile : File.T := File_NullFile;

procedure ValidateFile (Success : out Boolean)
  --# global in out ConfigFile;
  --# derives Success, ConfigFile from ConfigFile;
is
  OK : Boolean;

  procedure ReadPartitionsNumberValue
    --# global in out ConfigFile;
    --# out Success;
    --# derives ConfigFile,
    --# Success from ConfigFile;
is
      PartitionsNumber : Integer;
begin
  File_GetInteger(ConfigFile, PartitionsNumber, 0, Success); 
  if Success and then 
    (PartitionsNumber = DefaultValues.Number_Of_Partitions) then 
      Success := True; 
  else 
    Success := False; 
  end if;
  if File_EndOfFile(ConfigFile) then
    File_SkipLine(ConfigFile, 1);
  else 
```
Success := False;
end if;
end ReadPartitionsNumberValue;

procedure ReadSystemPartitionsNumber
--# global in out ConfigFile;
--# out Success;
--# derives ConfigFile,
--# Success from ConfigFile;
is
TheTitle : PartitionsNumberTextT;
Stop : Natural;
begin
File.GetString(ConfigFile, TheTitle, Stop);
if Stop = TheTitle'Last and then
TheTitle = PartitionsNumberTitle then
ReadPartitionsNumberValue;
else
Success := False;
end if;
end ReadSystemPartitionsNumber;

begin
FileSetName(TheFile => ConfigFile,
TheName => DefaultValues.Name_Of_Configuration_File);
if File.Exists(ConfigFile) then
File.OpenRead(TheFile => ConfigFile,
Success => Status);
if Success then
ReadSystemPartitionsNumber;
end if;
File.Close(TheFile => ConfigFile,
Success => OK);
Success := Success and OK;
else
Success := False;
end if;
end ValidateFile;

procedure Open(OK : out Boolean)
--# global in out ConfigFile;
--# derives ConfigFile, OK from ConfigFile;
is
Status : Boolean;
begin
FileSetName(TheFile => ConfigFile,
TheName => DefaultValues.Name_Of_Configuration_File);
if File.Exists(ConfigFile) then
File.OpenRead(TheFile => ConfigFile,
Success => Status);
else
Status := False;
end if;
end if;
end Open;
procedure Close (OK : out Boolean)
--# global in out ConfigFile;
--# derives ConfigFile, OK from ConfigFile;
is
Status : Boolean;
begin
File.Close(TheFile => ConfigFile,
Success => Status);
OK := Status;
end Close;

procedure Read_PRT_Values (Success : out Boolean)
--# global out ID_Partition, Size_Of_Partition, Duration_Partition, ←
Mode_Partition, Processes_Of_Partition;
--# in out ConfigFile;
--# derives Success, ConfigFile from ConfigFile &
--# 1D_Partition, Size_Of_Partition, Duration_Partition, ←
Mode_Partition, Processes_Of_Partition from ConfigFile;
is
Process : PartitionTypes.ProcessesType;

procedure ReadPartitionTitle
--# global in out ConfigFile;
--# out Success;
--# derives ConfigFile,
--# Success from ConfigFile;
is
TheTitle : PartitionT;
Stop : Natural;
begins
File.GetString(ConfigFile, TheTitle, Stop);
if Stop = TheTitle'Last and then
TheTitle = PartitionTitle then
if File.EndOfLine(ConfigFile) then
File.SkipLine(ConfigFile, 1);
Success := True;
else
Success := False;
end if;
else
Success := False;
end if;
end ReadPartitionTitle;

procedure ReadPartitionID
--# global in out ConfigFile;
--# out Success;
--# out 1D_Partition;
--# derives ConfigFile, Success from ConfigFile &
```pascal
---# ID_Partition from ConfigFile;
is
TheTitle : PartitionIDT;
Stop : Natural;

procedure ReadPartitionIDValue
---# global in out ConfigFile;
---# out Success;
---# out ID_Partition;
---# derives ConfigFile, Success from ConfigFile &
---# ID_Partition from ConfigFile;

    Value : Integer;
begin
    File.GetInteger(ConfigFile, Value, 0, Success);
    if Success and Value > 0 and Value <= $
    DefaultValues.Number_Of_Partitions then
        ID_Partition := Value;
    else
        ID_Partition := 1;
    end if;
    if File.EndOfLine(ConfigFile) then
        File.SkipLine(ConfigFile, 1);
    else
        Success := False;
    end if;
end ReadPartitionIDValue;

begin
    File.GetString(ConfigFile, TheTitle, Stop);
    if Stop = TheTitle.Last and then
        TheTitle = PartitionIDTitle then
            ReadPartitionIDValue;
    else
        Success := False;
        ID_Partition := 1;
    end if;
end ReadPartitionID;

procedure ReadPartitionSize
---# global in out ConfigFile;
---# out Success;
---# out Size_Of_Partition;
---# derives ConfigFile, Success from ConfigFile &
---# Size_Of_Partition from ConfigFile;

is
TheTitle : PartitionSIZE1;
Stop : Natural;

procedure ReadPartitionSIZEValue
---# global in out ConfigFile;
---# out Success;
---# out Size_Of_Partition;
---# derives ConfigFile, Success from ConfigFile &
```
Appendix C. SPARK packages

```plaintext
--- Size_Of_Parition from ConfigFile;

is
  Value : Integer;
begin
  File.GetInteger(ConfigFile, Value, 0, Success);
  if Success and Value in PartitionTypes.PartitionSizeAllowValues ←
  then
    Size_Of_Parition := Value;
  else
    Size_Of_Parition := PartitionTypes.PartitionSizeAllowValues'First;
  end if;
  if File.EndOfLine(ConfigFile) then
    File.SkipLine(ConfigFile, 1);
  else
    Success := False;
  end if;
end ReadPartitionSIZEValue;

begin
  File.GetString(ConfigFile, TheTitle, Stop);
  if Stop = TheTitle'Last and then
    TheTitle = PartitionSIZETitle then
      ReadPartitionSIZEValue:
  else
    Size_Of_Parition := PartitionTypes.PartitionSizeAllowValues'First;
    Success := False;
  end if;
end ReadPartitionSize;

procedure ReadPartitionDuration
  --- global in out ConfigFile;
  --- out Success;
  --- derives ConfigFile, Success from ConfigFile &
  --- Duration_Partition from ConfigFile;

  is
  TheTitle : PartitionDURATIONT;
  Stop : Natural;

procedure ReadPartitionDurationValue
  --- global in out ConfigFile;
  --- out Duration_Partition;
  --- derives ConfigFile, Success from ConfigFile &
  --- Duration_Partition from ConfigFile;

  is
  Value : Integer;
begin
  File.GetInteger(ConfigFile, Value, 0, Success);
  if Success and Value in PartitionTypes.Partition_Duration then
    Duration_Partition := Value;
  else
    Duration_Partition := 1;
```
end if;
if File.EndOfLine(ConfigFile) then
    File.SkipLine(ConfigFile, 1);
else
    Success := False;
end if;

begin
    File.GetString(ConfigFile, TheTitle, Stop);
    if Stop = TheTitle'Last and then
        TheTitle = PartitionDURATIONTitle then
        ReadPartitionDURATIONValue;
    else
        Duration_Partition := 1;
        Success := False;
    end if;
end ReadPartitionDURATION;

procedure ReadProcessTitle
--# global in out ConfigFile;
--# out Success;
--# derives ConfigFile,
--# Success from ConfigFile;
is
    TheTitle : ProcessT;
    Stop : Natural;
begin
    File.GetString(ConfigFile, TheTitle, Stop);
    if Stop = TheTitle'Last and then
        TheTitle = ProcessTitle then
            if File.EndOfLine(ConfigFile) then
                File.SkipLine(ConfigFile, 1);
                Success := True;
            else
                Success := False;
            end if;
        else
            Success := False;
        end if;
    else
        Success := False;
    end if;
end ReadProcessTitle;

procedure ReadProcess
--# global in out ConfigFile;
--# out Success;
--# out Process;
--# derives ConfigFile, Success from ConfigFile &
--# Process from ConfigFile;
is

procedure ReadProcessOperation
--# global in out ConfigFile;
--# out Success;
--# in out Process;

Appendix C. SPARK packages

--- derives ConfigFile, Success from ConfigFile &
--- Process from Process, ConfigFile;

is

TheTitle : ProcessOperationT;
RawOperation : OperationTextT;
Stop : Natural;
Matched : Boolean := False;

begin

Process.operation := PartitionTypes.OperationType'First;
File.GetString(ConfigFile, TheTitle, Stop);
if Stop = TheTitle'Last then
  TheTitle = ProcessOperationTitle
end if;

File.GetString(ConfigFile, RawOperation, Stop);
for OP in PartitionTypes.OperationType loop
  -- assert OP in PartitionTypes.OperationType;
  if Stop = OperationStringLookUp(OP).Length then
    -- assert OP in PartitionTypes.OperationType and
    -- Stop in OperationText1;
    Matched := True;
    for I in OperationText1 range 1 .. Stop loop
      -- assert OP in PartitionTypes.OperationType and
      -- Stop in OperationText1 and
      -- I in OperationText1 and
      -- I <= Stop;
      if OperationStringLookUp(OP).Text(I) /= RawOperation(I) then
        Matched := False;
        exit;
      end if;
    end loop;
    if Matched then
      Process.operation := OP;
      exit;
    end if;
  end loop;
  Success := Matched;
else
  Success := False;
end if;
end ReadProcessOperation;

procedure ReadProcessCommunicationMode
--- global in out ConfigFile;
--- out Success;
--- in out Process;
--- derives ConfigFile, Success from ConfigFile &
--- Process from Process, ConfigFile;

is

TheTitle : CommunicationModeT;
RawMode : CommunicationModeTextT;
Stop : Natural;
Matched : Boolean := False;
begin
Process.mode := PartitionTypes.READ;
File.GetString(ConfigFile, TheTitle, Stop);
if Stop = TheTitle’Last and then
  TheTitle = CommunicationModeTitle then
    File.GetLine(ConfigFile, RawMode, Stop);
  for M in PartitionTypes.CommunicationMode loop
  --# assert M in PartitionTypes.CommunicationMode;
    if Stop = CommunicationModeStringLookUp(M).Length then
      --# assert M in PartitionTypes.CommunicationMode and
      --# Stop in CommunicationModeTextI;
      Matched := True;
    for I in CommunicationModeTextI range 1 .. Stop loop
      --# assert M in PartitionTypes.CommunicationMode and
      --# Stop in CommunicationModeTextI and
      --# Stop = StopI and
      --# I in CommunicationModeTextI and
      --# I <= Stop;
      if CommunicationModeStringLookUp(M).Text(I) /= ←
        RawMode(I) then
        Matched := False;
        exit;
        end if;
    end loop;
    if Matched then
      Process.mode := M;
      exit;
    end if;
  end loop;
  Success := Matched;
else
  Success := False;
end if;
end ReadProcessCommunicationMode;
procedure ReadProcessCommunicationSize
  --# global in out ConfigFile;
  --# out Success;
  --# in out Process;
  --# derives ConfigFile, Success from ConfigFile &
  --# Process from Process, ConfigFile;
is
  TheTitle : CommunicationSizeT;
  Stop : Natural;

procedure ReadProcessCommunicationSizeValue
  --# global in out ConfigFile;
  --# out Success;
  --# in out Process;
  --# derives ConfigFile, Success from ConfigFile &
Appendix C. SPARK packages

```plaintext
--# Process from Process, ConfigFile;

is
Value : Integer;

begin
Process.size := 2;
File.getInteger(ConfigFile, Value, 0, Success);
if Success and Value in PartitionTypes.CommunicationsBlockSizeAllowValues then
    Process.size := Value;
end if;
if File.EndOfLine(ConfigFile) then
    File.SkipLine(ConfigFile, 1);
else
    Success := False;
end if;
end ReadProcessCommunicationSizeValue;

begin
Process.size := 2;
File.getString(ConfigFile, TheTitle, Stop);
if Stop = TheTitle'Last and then
    TheTitle = CommunicationSizeTitle then
        ReadProcessCommunicationSizeValue;
else
    Success := False;
end if;
end ReadProcessCommunicationSize;

procedure ReadProcessCommunicationTo
--# global in out ConfigFile;
--# out Success;
--# in out Process;
--# derives ConfigFile, Success from ConfigFile &
--#   Process from Process, ConfigFile;

is
TheTitle : CommunicationToT;
Stop : Natural;

procedure ReadProcessCommunicationToValue
--# global in out ConfigFile;
--# out Success;
--# in out Process;
--# derives ConfigFile, Success from ConfigFile &
--#   Process from Process, ConfigFile;

is
Value : Integer;

begin
Process.to := PartitionTypes.PartitionsNumber'First;
File.getInteger(ConfigFile, Value, 0, Success);
if Success and Value > 0 and Value <= DefaultValues.Number_Of_Partitions then
    Process.to := Value;
end if;
if File.EndOfLine(ConfigFile) then
```

---
Appendix C. SPARK packages

    File.SkipLine(ConfigFile, 1);
    else
        Success := False;
    end if;
end ReadProcessCommunicationToValue;

begin
    Process.to := PartitionTypes.PartitionsNumber 'First;
    File.GetString(ConfigFile, TheTitle, Stop);
    if Stop = TheTitle 'Last and then
        TheTitle = CommunicationToTitle then
            ReadProcessCommunicationToValue;
        else
            Success := False;
        end if;
end ReadProcessCommunicationTo;

begin
    Process.operation := PartitionTypes.OP_Communication;
    Process.blk.init := 1;
    Process.blk.sz := 1;
    Process.mode := PartitionTypes.READ;
    Process.size := 2;
    Process.to := PartitionTypes.PartitionsNumber 'First;
    ReadProcessOperation;
    if Process.operation = PartitionTypes.OP_Communication then
        ReadProcessCommunicationMode;
        if Success then
            ReadProcessCommunicationSize;
        end if;
    end if;
end ReadProcess;

begin
    ID_Parition := 1;
    Size_Of_Parition := PartitionTypes.PartitionSizeAllowValues 'First;
    Duration_Parition := 1;
    Mode_Parition := PartitionTypes.IDLE;
    Process.operation := PartitionTypes.OP_Nothing;
    Process.blk.init := 1;
    Process.blk.sz := 1;
    Process.mode := PartitionTypes.READ;
    Process.size := 2;
    Process.to := PartitionTypes.PartitionsNumber 'First;
    ReadPartitionTitle;
if Success then
  ReadPartitionID;
  --> assert True;
if Success then
  ReadPartitionSize;
  end if;
  --> assert True;
if Success then
  ReadPartitionDuration;
  end if;
if Success then
  for i in PartitionTypes.Index_Range_Processes_Per_Partititon loop
    --> assert i in <-
      PartitionTypes.Index_Range_Processes_Per_Partititon;
    ReadProcessTitle;
    --> assert True;
    if Success then
      ReadProcess;
      end if;
      --> assert True;
      Processes_Of_Partition(i) := Process;
    end loop;
    --> assert True;
    if Success then
      Mode_Partition := PartitionTypes.IDLE;
      end if;
    end if;
  end Read_PRT_Values;

function Get_Partition_ID return PartitionTypes.IdType
  --> global ID_Partition;
is
begin
  return ID_Partition;
end Get_Partition_ID;

function Get_Partition_Size return PartitionTypes.PartitionSizeAllowValues
  --> global Size_Of_Partition;
is
begin
  return Size_Of_Partition;
end Get_Partition_Size;

function Get_Partition_Duration return PartitionTypes.Partition_Duration
  --> global Duration_Partition;
is
begin
  return Duration_Partition;
end Get_Partition_Duration;

function Get_Partition_Mode return PartitionTypes.Partition_Mode
  --> global Mode_Partition;
Appendix C. SPARK packages

681 is
682 begin
683 return Mode_Partition;
684 end Get_Partition_Mode;
685
function Get_Partition_Processes return PartitionTypes.Processes_List
686 --# global Processes_Of_Partition;
687 is
688 begin
689 return Processes_Of_Partition;
690 end Get_Partition_Processes;
691
692 procedure Read_Size_Of_Blocks_For_Communication (Success : out Boolean)
693 --# global in out Size_Of_Blocks_For_Communication;
694 --# in out ConfigFile;
695 --# derives Success, ConfigFile from ConfigFile &
696 --# Size_Of_Blocks_For_Comunication from ←
697 Size_Of_Blocks_For_Communication, ConfigFile;
698 is
699 procedure SetDefaults
700 --# global out Size_Of_Blocks_For_Communication;
701 --# derives Size_Of_Blocks_For_Communication from ;
702 is
703 begin
704 Size_Of_Blocks_For_Communication := 4;
705 end SetDefaults;
706
707 procedure ReadCommunicationBlocksSizeValue
708 --# global in out ConfigFile;
709 --# out Success;
710 --# in out Size_Of_Blocks_For_Communication;
711 --# derives ConfigFile,
712 --# Success from ConfigFile &
713 --# Size_Of_Blocks_For_Comunication from ←
714 Size_Of_Blocks_For_Communication, ConfigFile;
715 is
716 Value : Integer;
717 begin
718 File.GetInt( ConfigFile, Value, 0, Success);
719 if Success and then
720 (Value >= 1 and Value <= ←
721 DefaultValues.CommunicationsBlockSizeMaxValue) then
722 Success := True;
723 Size_Of_Blocks_For_Communication := Value;
724 else
725 Success := False;
726 end if;
727 if File.EndOfFile(ConfigFile) then
728 File.SkipLine(ConfigFile, 1);
729 File.SkipLine(ConfigFile, 1);
730 else
731 Success := False;
end if;
end ReadCommunicationBlocksSizeValue;

procedure ReadBlockSize
--$ global in out ConfigFile;
--$ out Success;
--$ derives Success, ConfigFile from ConfigFile;
is
  TheTitle : CommunicationBlocksSizeT;
  Stop : Natural;
begin
  File.SkipLine(ConfigFile, 1);
  File.GetString(ConfigFile, TheTitle, Stop);
  if Stop = TheTitle'Last and then
    TheTitle = CommunicationBlocksSizeTitle then
      Success := True;
  else
    Success := False;
  end if;
end ReadBlockSize;
begin
  ReadBlockSize;
  if Success then
    ReadCommunicationBlocksSizeValue;
  else
    SetDefaults;
  end if;
  Success := Success;
end ReadSizeOfBlocksForCommunication;

function GetSizeOfBlocksForCommunication return PartitionTypes.CommunicationsBlockSizeAllowValues is
begin
  return SizeOfBlocksForCommunication;
end GetSizeOfBlocksForCommunication;

procedure ReadPIFP(Success : out Boolean)
--$ global in out ConfigFile;
--$ derives Success, ConfigFile from ConfigFile;
is
  TheTitle : PifpTextT;
  Stop : Natural;
begin
  File.GetString(ConfigFile, TheTitle, Stop);
  if Stop = TheTitle'Last and then
    TheTitle = PifpTitle then
      Success := True;
      File.SkipLine(ConfigFile, 1);
    else
      Success := False;
  end if;
Appendix C. SPARK packages

```plaintext
end Read_PIFP;

procedure Read_PIFP_Line(Success : out Boolean; ID : out PartitionTypes.PartitionsNumber)
  --# global in out PIFP_Line;
  --# in out ConfigFile;
  --# derives PIFP_Line from PIFP_Line, ConfigFile &
  --# ID from ConfigFile &
  --# Success, ConfigFile from ConfigFile;
is
  PID_Temp : PartitionTypes.PartitionsNumber;
  Mode_Temp : TablesTypes.FlowMode;

procedure Read_ID
  --# global out Success;
  --# out ID;
  --# in out ConfigFile;
  --# derives Success, ID, ConfigFile from ConfigFile;
is
  TheTitle : IDTextT;
  Stop : Natural;

procedure Read_ID_Value
  --# global out Success;
  --# out ID;
  --# in out ConfigFile;
  --# derives Success, ID, ConfigFile from ConfigFile;
is
  Value : Integer;

begin
  ID := PartitionTypes.PartitionsNumber 'First;
  File.GetInteger(ConfigFile, Value, 0, Success);
  if Success and then
    (Value in PartitionTypes.PartitionsNumber) then
      Success := True;
      ID := Value;
  else
    Success := False;
  end if;
  if File.EndOfLine(ConfigFile) then
    File.SkipLine(ConfigFile, 1);
  else
    Success := False;
  end if;
end Read_ID_Value;

begin
  ID := PartitionTypes.PartitionsNumber 'First;
  File.GetString(ConfigFile, TheTitle, Stop);
  if Stop = TheTitle 'Last and then
    TheTitle = IDTitle then
      Read_ID_Value;
  else
    Success := False;
  end if;
end Read_ID_Value;
```
procedure Read_ID (P_ID : out PartitionTypes.PartitionsNumber; Mode : out TablesTypes.FlowMode)
  --# global in out ConfigFile;
  --# derives Success, P_ID, Mode, ConfigFile from ConfigFile;
is
  Value : Integer;

procedure Read_Mode
  --# global in out ConfigFile;
  --# out Success;
  --# out Mode;
  --# derives ConfigFile, Success from ConfigFile &
  --# Mode from ConfigFile;
is
  RawMode : FlowTextT;
  Stop : Natural;
  Matched : Boolean := False;
begin
  Mode := TablesTypes.FlowMode’Last;
  File.ReadLine(ConfigFile, RawMode, Stop);
  for M in TablesTypes.FlowMode loop
    --# assert M in TablesTypes.FlowMode;
    if Stop = FlowStringLookUp(M).Length then
      --# assert M in TablesTypes.FlowMode and
      --# Stop in FlowTextI;
      Matched := True;
      for I in FlowTextI range 1 .. Stop loop
        --# assert M in TablesTypes.FlowMode and
        --# Stop in FlowTextI and
        --# Stop = Stop % and
        --# I in FlowTextI and
        --# I <= Stop;
        if FlowStringLookUp(M).Text(I) /= RawMode(I) then
          Matched := False;
          exit;
        end if;
      end loop;
    end if;
  end loop;
end if;
if Matched then
  Mode := M;
  exit;
end if;
end Read_Mode;
begin
  P_ID := PartitionTypes.PartitionsNumber’First;
  Mode := TablesTypes.FlowMode’Last;
  File.GetInteger(ConfigFile, Value, 0, Success);
  if Success then
    (Value in PartitionTypes.PartitionsNumber) then

P_ID := Value;
Read_Mode;
else
    Success := False;
end if;
end Read_Flow;

procedure SetDefaults
--# global in out PIFP_Line;
--# derives PIFP_Line from PIFP_Line;
is
begin
    for i in PartitionTypes.PartitionsNumber loop
        --# assert i in PartitionTypes.PartitionsNumber;
        PIFP_Line(i) := TablesTypes.W;
    end loop;
    end SetDefaults;
begin
    Read_ID;
if Success then
    for i in PartitionTypes.PartitionsNumber loop
        --# assert i in PartitionTypes.PartitionsNumber;
        Read_Flow(P_ID.Temp, Mode.Temp);
        PIFP_Line(P_ID.Temp) := Mode.Temp;
    end loop;
else
    SetDefaults;
end if;
end Read_PIFP_Line;

procedure Read_PRT_Exec_Sequence(Success : out Boolean)
--# global out Partitions_Execution_Sequence;
--# in out ConfigFile;
--# derives Partitions_Execution_Sequence from ConfigFile &
--# Success, ConfigFile from ConfigFile ;
is
    PartitionID : PartitionTypes.PartitionsNumber;
procedure Read_PartitionsExecutionNumber
--# global in out ConfigFile;
--# out Success;
--# derives Success, ConfigFile from ConfigFile ;
is
    TheTitle : PartitionsExecutionNumberT;
    Stop : Natural;
    temp : Integer;
begin
    File.GetString(ConfigFile, TheTitle, Stop);
if Stop = TheTitle.'Last and then
    TheTitle = PartitionsExecutionNumberTitle then
        File.GetInteger(ConfigFile, temp, 0, Success);
if Success and then
    (temp = DefaultValues.PartitionsExecutionNumber) then
Success := True;
else
  Success := False;
end if;
if File.EndOfLine(ConfigFile) then
  File.SkipLine(ConfigFile, 1);
else
  Success := False;
end if;
else
  Success := False;
end if;
end

procedure Read_PartitionsExecutionNumber
-* global in out ConfigFile;*
-* out Success;*
-* derives Success, ConfigFile from ConfigFile;*
is
  TheTitle : PartitionsExecutionSequenceT;
  Stop : Natural;
begin
  File.GetString(ConfigFile, TheTitle, Stop);
  if Stop = TheTitle'Last and then
    TheTitle = PartitionsExecutionSequenceTitle then
      if File.EndOfLine(ConfigFile) then
        File.SkipLine(ConfigFile, 1);
        Success := True;
      else
        Success := False;
      end if;
    else
      Success := False;
    end if;
  end if;
end Read_PartitionsExecutionNumber;

procedure Read_PartitionsExecutionSequenceTitle
-* global in out ConfigFile;*
-* out Success;*
-* derives Success, ConfigFile from ConfigFile;*
is
  TheTitle : PartitionsExecutionSequenceT;
  Stop : Natural;
begin
  File.GetString(ConfigFile, TheTitle, Stop);
  if Stop = TheTitle'Last and then
    TheTitle = PartitionsExecutionSequenceTitle then
      if File.EndOfLine(ConfigFile) then
        File.SkipLine(ConfigFile, 1);
        Success := True;
      else
        Success := False;
      end if;
    else
      Success := False;
    end if;
  end if;
end Read_PartitionsExecutionSequenceTitle;

procedure Read_PartitionValue
-* global in out ConfigFile;*
-* out Success;*
-* in out PartitionID;*
-* derives ConfigFile,*
-* derives Success from ConfigFile &*
-* PartitionID from PartitionID, ConfigFile;*
is
  Value : Integer;
begin
  File.GetInteger(ConfigFile, Value, 0, Success);
  if Success and then
    (Value in PartitionTypes.PartitionsNumber) then
      Success := True;
      PartitionID := Value;
    else
      Success := False;
    end if;
end Read_PartitionValue;
Appendix C. SPARK packages

```plaintext
if File.EndOfLine(ConfigFile) then
    File.SkipLine(ConfigFile, 1);
else
    Success := False;
end if;
end ReadPartitionValue;

begin
    Partitions_Execution_Sequence := TableTypes.PartitionsExecutionSequenceTable (TableTypes.PartitionsExecutionIndex = First);
    Read_PartitionsExecutionNumber;
    if Success then
        Read_PartitionsExecutionSequenceTitle;
        if Success then
            for i in TableTypes.PartitionsExecutionIndex loop
                PartitionID := TableTypes.PartitionsNumber.First;
                ReadPartitionValue;
                if Success then
                    Partitions_Execution_Sequence(i) := PartitionID;
                end if;
            end loop;
        end if;
    end if;
end Read_PRT_Exec_Sequence;
end ConfigValues;
```

Code Listing C.17: ConfigValues body package
Appendix D

Example of a configuration file

An example of a valid configuration file is presented in this appendix. In this configuration file, as we can see in the first three lines, the kernel has 3 partitions, it has 20 blocks of memory for communication, and every one is divided in 4. Each partition has obligatorily 2 processes, as we can see in line number four.

The partition with ID=1, as we can see in between lines twenty and twenty-nine, has size=350, has a duration=12, and has one process that wants to communicate with the partition with ID=2, to “Write” and the message size is 5. It has another process to do “Nothing”. The partition with ID=2, as we can see in between lines six and eighteen, has size=149, has a duration=3 and has two processes that want to communicate; both want to communicate with partition ID=1, one is to “Read” being the message size 4, and the other is to “Write” being the message size 1. Finally, the partition with ID=3, as we can see in between lines thirty-one and thirty-seven, has size=170, has a duration=11 and both processes are specified to do “Nothing”.

The cycle of execution consists of 4 partitions, and the sequence is respectively, ID=2, ID=1, ID=2, ID=3, as we can see in between lines thirty-eight and forty-three. The partition information flow policy, as we can see from line forty-five until de end of the file, is composed by: the partition ID=1, can communicate with partitions ID=2 and ID=3 to “Read/Write”; the partition which ID=2 only can communicate with partition ID=1 to “Read/Write”; and the partition ID=3 can not communicate with any of them.
Appendix D. Example of a configuration file

```
PartitionsNumber 3
NumberOfCommunicationsBlocks 20
CommunicationBlocksSize 4
NumberOfProcessesPerPartition 2
PARTITION
PartitionID 2
PartitionSIZE 149
PartitionDURATION 3
PROCESS
ProcessOperation Communication
CommunicationMode Read
CommunicationSize 4
CommunicationTo 1
PROCESS
ProcessOperation Communication
CommunicationMode Write
CommunicationSize 1
CommunicationTo 1
PARTITION
PartitionID 1
PartitionSIZE 350
PartitionDURATION 12
PROCESS
ProcessOperation Communication
CommunicationMode Write
CommunicationSize 5
CommunicationTo 2
PROCESS
ProcessOperation Nothing
PARTITION
PartitionID 3
PartitionSIZE 170
PartitionDURATION 11
PROCESS
ProcessOperation Nothing
PROCESS
ProcessOperation Nothing
PartitionsExecutionNumber 4
PartitionsExecutionSequence
2
1
2
3
PIFP
ID 3
1 N
2 N
ID 1
2 RW
3 RW
ID 2
1 RW
3 N
```