UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Towards a Formally Verified Space Mission Software using SPARK

Paulo Miguel Ferreira Neto

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Prof. Doutor Simão Melo de Sousa

Covilhã, junho de 2019

Acknowledges

I would like to express my gratitude to the persons who directly or indirectly made the conclusion of this dissertation a true reality.

To Professor Simão Melo de Sousa, for coordinating this dissertation. Thank you for all the time spent on scientific guidance and support, constant feedback and for enabling the possibility of doing this dissertation in industrial context at Critical Software, SA.

Also many thanks for Joaquim Tojal for being a “father” to me on Critical Software, SA. Thank you for the help on my integration at the company and also for the constant guidance, support and feedback.

I would also want to thank to my colleagues that I worked in Critical Software, SA. with a special thanks to José Veríssimo that was very important for my integration in the ExoMars project.

Now to my parents, Paula and Paulo, and my sister, Liliana, for always being present in the most stressful situations turning them in unforgettable moments and thank you for always trusting in me.

To my girlfriend, Andreia, thank you for the patience, love and care in the worst moments, not only during this year but for the past five years in my academic course.

Last, but not least, to the family that Covilhã gave to me. Thank you Édi, Emanuel, Guilherme, José, Nelson and Raul for the friendship and for the best and the worst moments in the past five years. A special thanks for Édi and José for their hospitality in my journeys to Covilhã during this year.

Thank you all!!

Resumo

Este trabalho de investigação está inserido no âmbito da dissertação do curso de Mestrado em Engenharia Informática da Universidade da Beira Interior realizado em contexto industrial na empresa Critical Software, SA.

Este documento apresenta o projeto de verificação e validação formal do *Software* Central do ExoMars Trace Gas Orbiter que foi totalmente implementado pela Critical Software em parceria com a Thales Alenia Space, para uma missão da Agência Espacial Europeia direcionada à exploração da atmosfera Marciana. O trabalho realizado tratou da reformulação do código implementado em Ada para SPARK com o objetivo de avaliar as suas capacidades de análise de código e as técnicas de *Design-by-Contract* para validação e verificação formal do *software* de voo de um satélite.

Em sistemas *Safety Mission-Critical* a importância de um *software* seguro e confiável conduz à introdução novos métodos de desenvolvimento com grande integridade como uma alternativa aos testes de *software* eficiente e de baixo custo. Tal como Edger Dijsktra disse, testar *software* apenas deteta a presença de erros mas não prova a sua ausência e, para além disso, a deteção é tardia. Existe uma cresente necessidade de assegurar as propriedades funcionais e de *safety* do sistema na fase de pré-produção.

A abordagem *Design-by-Contract* é um tipo de verificação formal de código, baseada na Lógica de Hoare, que pode reduzir os custos de duas formas, porque dá-nos elevadas garantias de que o código está livre de erros de execução e que irá executar como pretendido, logo na fase de desenvolvimento onde a correção de erros é menos custosa. A ferramenta SPARK é usada para realizar a verificação formal de código para a linguagem de programação Ada, usado na maioria dos casos para o desenvolvimento de sistemas *Safety Mission-Critical*. A análise do SPARK é direcionada para a garantia de um correto fluxo de informação e o correto comportamente na execução dos programas.

Palavras-chave

Métodos Formais, *Design-by-Contract*, Verificação Formal de Código, Análise de Fluxo, Sistemas *Safety Mission-Critical*, ExoMars

Resumo alargado

Uma falha num sistema *Safety Mission Critical* pode resultar na perda de vidas humanas, ferimentos, danos ambientais, incumprimentos dos objetivos da missão e ainda grandes perdas económicas. Existe uma necessidade acrescida de assegurar propriedades como *Reliability*, *Safety* e *Security* para que um sistema seja capaz de evitar todas estas possíveis consequências. *Reliability*, ou Confiabilidade, está diretamente relacionada com a confiança que temos dos sistemas e trata-se da probabilidade de um sistema estar operacional durante um período de tempo específico e, *Safety* e *Security* estão completamente ligadas à confiança que está associada ao sistema implementado. *Safety* e *Security* têm a mesma tradução para a língua portuguesa mas, têm objetivos distintos para as propriedades que este tipo de sistemas requerem. Enquanto que *Safety* é uma propriedade que garante que o sistema irá operar sem falhas, *Security* tenta garantir que o sistema está protegido de eventuais ameaças que possam surgir de componentes externas ao sistema.

As consequências das falhas mencionadas anteriormente traduzem-se em elevados custos que levam à introdução de metodologias ágeis com o intuito de assegurar as propriedades criticas para a confiança no sistema. Estas metodologias são também elas dispendiosas muito devido ao seu grau de complexidade e, normalmente, são descartadas dos processos de desenvolvimento de sistemas que não necessitam de um selo de qualidade tão vincado por não se tratarem de técnicas rentáveis para os objetivos definidos para esses sistemas.

Nas empresas inseridas na área do desenvolvimento de *software* de sistemas críticos, existe uma necessidade normativa de conhecimento e aplicação de técnicas para verificação e validação formal de código tais como, *Model Checking*, *Theorem Proving* and *Design-by-Contract*. Na Critical Software, SA., a técnica frequentemente mais utilizada até ao momento para verificação e validação do código deste tipo de sistemas passa pela realização de testes de *software*. Uma técnica que por vezes se pode tornar dispendiosa devido ao processo tardio de deteção de erros. O objetivo deste trabalho passa por estudar uma metodologia que possa vir a ser implementada no processo de desenvolvimento de sistemas *Safety Mission Critical* de forma a reduzir substancialmente alguns dos custos obtidos com a realização de testes de software. A técnica a ser aplicada, *Design-by-Contract*, trata-se de uma metodologia de verificação formal de código normalmente realizada por ferramentas de análise estática de código que irá permitir a deteção dos erros na fase de implementação onde a sua correção será menos custosa e, assim, reduzir parcialmente as práticas de testes de *software*.

Com este objetivo em mente, o trabalho desenvolvido ao longo desta investigação passou por testar a ferramenta SPARK normalmente usada para a verificação formal de código Ada usando a abordagem de *Design-by-Contract*. O código aqui verificado pertencia ao *software* central do ExoMars Trace Gas Orbiter (TGO) que faz parte de uma missão da European Space Agency (ESA) para a exploração de atmosfera Marciana com o objetivo principal medir os seus níveis de metano. O *software* central do ExoMars TGO foi totalmente implementado na linguagem Ada pela Critical Software, SA. em parceria com Thales Alenia Space.

A abordagem *Design-by-Contract* é baseada na Lógica de Hoare, que defende a definição da problemática de um programa em formulas matemáticas de lógica de primeira ordem que pos-

teriormente sofrem a aplicação de alguns axiomas e regras de inferência para finalmente ser provada a validade dessa fórmula com provadores automáticos. Esta abordagem defende a especificação de um contrato entre o programa em análise e o seu chamador. O chamador tem de satisfazer uma condição inicial, *Pre-Condition*, para que o programa possa ser executado e, no fim, o programa em análise tem de satisfazer uma condição, *Post-Condition*, que verificada garante a correção do programa. Muitas das vezes, a prova das *Post-Conditions* não é uma tarefa trivial e para isso existe outro tipo de contratos, Asserções, que devem ser sempre válidas para conseguir garantir a consistência do corpo do programa como, código condicional ou execução de ciclos.

O SPARK é uma linguagem que permite a verificação formal do código Ada usando a abordagem de *Design-by-Contract*. A ferramenta permite anotar o código Ada com os contratos já mencionados na sua sintaxe definida e posteriormente utiliza a sua ferramenta de verificação que, com a auxilio de provadores externos, tenta provar a integridade dos programas. A versão mais recente, SPARK 2014, integrou uma nova ferramenta de verificação que realiza a análise estática, o *GNATProve*. O GNATProve por sua vez é auxiliado pelo Why3 na prova dos programas, traduzindo as anotações para WhyML e provando com provadores como CVC4, Z3 e Alt-Ergo. O trabalho do SPARK divide-se em duas análises, a Análise de Fluxo e a Verificação do Código.

As principais preocupações da **Análise de Fluxo** é os dados, detetando casos que possam levantar possíveis exceções durante a execução, como uso de ponteiros, casos de *alias* e efeitos colaterais ao comportamento requerido para as funções. Deteta automaticamente se as variáveis e objetos são corretamente declarados, instanciados e atualizados. E ainda prova o correto fluxo da informação entre os programas recorrendo a contratos especiais que permitem definir o uso de variáveis e objetos globais e ainda as suas dependências.

A **Verificação do Código** é a análise que prova a integridade do programa. Primeiro, mais uma vez de forma automática, o SPARK deteta a possível ocorrência de erros que possam ocorrer durante a execução do programa tais como, divisões por zero, possíveis *overflows*, violação dos intervalos definidos para determinado tipo de dados, entre outros. Existem duas possibilidades para um *warning* proveniente desta análise, ou o código está realmente errado e tem de ser alterado ou o SPARK lançou um falso alarme que pode facilmente ser provado com o uso dos contratos para definir certas propriedades que não sejam bem conhecidas pela ferramenta de análise. Segundo, o SPARK, recorrendo aos contratos anotados e ao uso das as ferramentas que tem ao seu dispor tenta chegar a prova da correção funcional do programa, ou seja, se ele se comporta de acordo com as condições especificadas na *Post-Condition*.

Com o levantamento do estado da arte realizado foi possível concluir que o uso de SPARK na indústria tem vindo a crescer significativamente ao longo dos anos e a elevar os níveis de integridade dos sistemas. Tendo sido, na maioria dos casos, atingidos os níveis de integridade mais altos requisitados pelos *standards* mais rigorosos. No domínio espacial, o estado da arte relativo ao uso do SPARK ainda é bastante reduzido mas, os recentes resultados de um projeto académico que implementou o *software* de voo de um *CubeSat* (Satélite de 1 Unidade) em SPARK que levou a que este operasse sem falhas durante o período definido para a missão e outros casos de estudo que provam a possibilidade de construir o *software* de voo de um satélite com SPARK atingindo todos os requisitos definidos pelos *standards* da *European Cooperation Space Standardization*, têm sido um incentivo para a introdução de SPARK ou de outro tipo de ferramentas de verificação

de código em futuros projetos no domínio espacial.

No satélite ExoMars, tal como em outros sistemas de satélites, existem cinco sistemas diferentes mas para este caso de estudo apenas foi considerado o sistema de Regulação de Temperatura. Este sistema tem como função calcular as temperaturas médias dos componentes físicos do satélite e ativar ou desativar aquecedores para regular a temperatura de determinada componente, que pode discrepar entre valores bastante elevados ou bastante negativos dependendo da orientação da componente em relação ao sol ou sombra.

O sistema de Regulação de Temperatura realiza uma tarefa cíclica que regulariza a temperatura dos componentes de *hardware* constantemente. A abordagem apresentada nesta dissertação para a prova do código do sistema consistiu em, primeiro realizar a análise do SPARK em todas as componentes que a tarefa principal era dependente para, no final, ao realizar a análise desta termos a garantia de que tudo o que é externo está correto e é confiável.

Como o código não foi implementado com a análise do SPARK em mente, este continha funcionalidades que não são válidas para a análise do SPARK. Então, a abordagem inicial foi utilizar as mesmas funcionalidades, separando do SPARK as funções e/ou funcionalidades não compatíveis em interfaces externas bem assinaladas, sem influenciar o comportamento inicial do programa. Ultrapassado este problema foi realizada a análise para garantir um correto fluxo da informação durante a execução execução do sistema de Regulação de Temperatura, que o código está livre de erros que possam ocorrer durante a execução e que se comporta como requisitado.

Em termos de resultados obtidos podemos afirmar que o processo de verificação ilustrado aqui nesta dissertação veio acrescentar ainda mais um selo de qualidade às elevadas garantias que já tinham sido atribuídas ao *software* implementado com o processo de verificação e validação realizado pela Critical Software. Os resultados finais permitiram concluir que este trabalho veio assegurar um nível de qualidade do *software* equiparado aos níveis mais elevados definidos pelos standards DO-178B e IEC 61508. De salientar, que também foi possível desenhar um conjunto de recomendações de otimização para o código do sistema em análise.

Abstract

This research work is in the scope of the Master course in Computer Science of University of Beira Interior dissertation and introduces the research undertaken in a Master Thesis done in collaboration with Critical Software, SA.

This document presents the effort for verification and validation of ExoMars Trace Gas Orbiter Central Software that was fully implemented by Critical Software in cooperation with Thales Alenia Space, for an European Space Agency exploration of Mars atmosphere mission. The work was an Ada implemented code reformulation to SPARK aiming the capability evaluation of code analysis and *Design-by-Contract* techniques for validation and verification of a spacecraft on board software.

On *Safety Mission-Critical* systems the importance of secure and reliable software path to the introduction of new high integrity development methods as an efficient and low cost alternative of software testing. As Edger Dijsktra said, testing software only detect the presence of bugs but not prove his absence and, besides that, it is a late detection. There is an increased need of safety and functional properties assurance before the system deployment.

Design-by-Contract approach is a kind of formal code verification, based on Hoare Logic, that can reduce the costs in two ways, because it gave earlier high guarantees that code is free of run-time errors and run as expected, in the development phase where bugs correction is cheaper. SPARK toolset is used to perform formal code verification for Ada programming language, mostly used on *Safety Mission-Critical* systems development. SPARK analysis is targeted for assurance of correct information flow and the correct behaviour of programs execution.

Keywords

Formal Methods, Design-by-Contract, Formal Code Verification, Flow Analysis, Safety Mission-Critical Systems, ExoMars

Contents

List of Figures

List of Tables

Listings

Acronyms

ACSL ANSI/ISO C Specification Language

AJ Adjustable

ANSSI Agence Nationale de la Sécurité des Systèmes d'Information

APM Antenna Pointing Mechanism

AST Astrium Space Trasnportation

ATV Automated Transfer Vehicle

BSP Board Support Package

CA Certification Authority

CC Common Criteria

CSW Central SoftWare

CoRE Consortium Requirements Engineering

DAL Design Assurance Level

DSN Deep Space Network

EAL Evaluation Assurance Level

EDL Entry, Descent Landing

EDM Entry, Descent Module

ELaNa Educational Launch of Nano-satellites

ESA European Space Agency

FCV Functional Chain Validation

FDIR Failure Detection, Isolation and Recovery

GEONS Goddard Enhanced Onboard Naviagtion System

GNC Guidance, Navigation and Control

GPS Global Positioning System

HDSW Hardware Dependent Software

HK HouseKeeping

IDE Integrated Development Environment

IEC International Eletrotechnical Commissions

iFACTS Interim Future Area Control Tools Support

I/O Input/Output

JML Java Modeling Language

LASC LOCKHEED Aeronautical Systems Company

MAS Mission Application Software

MCD MULTOS Carrier Device

MTL Mission Timeline

MULTOS Multi-Application Operating System

MVM Mission and Vehicle Management

NASA National Aeronautics and Space Administration

NATS National Air Traffic Services

NSA National Security Agency

OBC On-Board Computer

OS Operating System

PCDU Power Conditioning and Distribution Unit

PUS Package Utilization Standard

RTK Real-Time Kernel

SADM Solar Array Drive Mechanism

SDS Software Design specification

SGS Solar Generation Subsystem

SHOLIS Ship Helicopter Operating Limits Information System

SIL Software Integrity Level

SMS System Management Software

SMT Satisfiability Modulo Theories

SMU Spacecraft Management Unit

SRS Software Requirements Specification

TAS-F Thales Alenia Space - France

TC Telecommand

TGO Trace Gas Orbiter

TIS Tokeneer ID Station

TM Telemetry

TR Thermal Regulation

UHF Ultra Hight Frequency

VC Verification Condition

Chapter 1

Introduction

This research work aims to evaluate the capability of code analysis using SPARK toolset and his *Design-by-Contract* techniques for validation and verification of a spacecraft on board software. This chapter introduces and defines the problem addressed to this research work. The first sections presents the motivation, context and scope to this research work and the next one expose the contributions resulting of this work and finally a description of the contents for each chapter of this dissertation.

1.1 Motivation

Systems such as medical devices, financial software, autonomous cars, air traffic control technology, railway control and space missions are categorized as Safety and Mission Critical systems. Systems that in case of failure, the consequences can cost humans life, damage the environment or have an impact on financial resources [6, 7, 8]. To note that when addressing space industry projects, the association is always to Mission Critical systems.

In this type of systems there are two components completely related which are software reliability and software quality. If we increase the quality by decreasing the probability of getting errors, one can argue that the system has more reliability guarantees [9]. In general, the most cases of errors on critical software development are due to requirement specification. Requirements not clear or ambiguous and insufficient knowledge of the formalisms between the systems team and software team and vice-versa [10]. Therefore, it is expected significant advances in specification, verification and validation of dependable Safety Mission-Critical systems [8].

Software properties verification is a costly and complex process that could be avoided by using formal program verification during the development process. Formal verification guarantees that the properties will always be verified, independently of the context. Thus, increasing quality and reliability. There are some standards that recommend the use of formal methods to produce systems with high quality and low defects, for instance, EN 20128 for the railway industry and IEC 61508 for industrial processes. In avionics industry, the standard DO-178C recommends the use of formal methods in pair with testing techniques such unit tests. Standards for domains such as automotive (ISO 26262), nuclear (IEC 60880) and space (ECSS-QST-80C) recommend the use of formal methods as verification technique [1]. The Common Criteria (CC) [11] Evaluation Assurance Level (EAL), a standard more rigorous and exigent, describe seven EAL (1 to 7) as a measure of assurance quality, defining the rigor that has been applied to assure the pretended security and safety properties of the system. On the highest levels (5, 6 and 7), corresponding to the strongest assurance level, are totally recommended the use of formal methods [12]. At the fifth level, EAL-5, formal methods are recommended on to requirements definition and semi-formal methods for functional specification and high-level design. The EAL-6 joins EAL-5 with semi-formal verification for low-level design. EAL-7 requires the use of formal methods in all stages mentioned at the previous levels.

Design-by-Contract technique is one of the approaches to perform formal verification. Here, contracts provide Boolean expressions to prove the complete absence of errors during run-time and if the implementation complies with the required specifications [13, 14]. These Boolean expressions (Pre- and Post-conditions) are imposed obligations to the sub-programs that shall be satisfied. Pre-Condition it is a condition that must be verified before the execution of some function and the Post-Condition it is another condition that needs to be verified after the execution.

For this research work, the SPARK toolset [15] was chosen to provide formal verification using the *Design-by-Contract* approach. SPARK is a subset of Ada language [16] and its most recent version (SPARK 2014) was designed to interpret Ada 2012 contracts. SPARK toolset translates Ada code to WhyML and the SPARK verifications to Why3 [17]. Proof obligations are processed with Why3 interface. Other specification toolsets rely on Why3 and its specification language WhyML. Toolsets such, Frama-C [18] and Java [19], with ANSI/ISO C Specification Language (ACSL) [20] and Java Modeling Language (JML) [21] as specification languages, respectively.

1.2 Context

This research work is part of a partnership between University of Beira Interior and a Portuguese company that operates globally on the targeted domains of this work. Moved by many years of excellent work, the aforementioned partnership continues to have the goal of bridging relevant research with real-world systems and software services for mission and business-critical applications. The strong communication between these two entities was important to the success of this work.

The research work aims the partial verification and validation of ExoMars TGO Central Soft-Ware (CSW) components that were already developed and deployed on space for an ESA mission to explore the Mars planet measuring the trace gas levels. The verification toolset chosen was SPARK an Ada subset designed to prove real-time embedded software where the maximum priority requirements are safety, security and reliability.

1.3 Objectives

This research work was divided in two principal stages:

- The first one, a more theoretical stage, was divided in other two:
  - The state of the art study, reading and interpreting a set of scientific articles related to the thematic of formal verification with SPARK on Safety Mission-Critical systems, essentially on space domain;
  - The tools study, starting from some Ada programming language trainings and after, training the formal verification toolset SPARK to apply on Ada code. From these trainings results a simple application for a STM32F4 micro-controller developed with Ada and verified with SPARK;

- The second one, a practical stage, was applying the SPARK toolset to the Ada code, produced by Critical Software, SA., of TGO CSW components. This stage followed the following steps:
  - Well understood of the required behaviour for the software;
  - Code Annotation, with SPARK contracts;
  - Prove the code correct behaviour and showing that SPARK can be a complement to software testing.

1.4 Contributions

The main contributions of this work are the following:

- Formal Specification and Verification of ExoMars TGO Thermal Regulation (TR) system;
- Integration of Formal Verification approach in Critical Software, SA. development of critical software;
- Technical Report about the project under submission.

1.5 Outline

This document is organized as follows:

- **Chapter 2** presents an overview of the *Design-by-Contract* formal verification approach and the logic which is based, Hoare Logic;
- **Chapter 3** presents an overview on Ada and SPARK tools and verification methodology;
- **Chapter 4** describes the state of the art related with the SPARK use on industrial context during the years;
- **Chapter 5** presents the ExoMars TGO mission and system architectures overview with special emphasis on the component analysed in this work;
- **Chapter 6** exposes the delail design of TR system;
- **Chapter 7** exposes the verification work for each TR component and presents the final results and the SPARK performance evaluation on this work;
- **Chapter 8** present the final conclusions and future improvements;

Chapter 2

Formal Methods

2.1 Introduction

As a complement to other sections in this document, this chapter addresses in detail the targeted formal verification technique. Will be introduced necessary aspects of *Design-by-Contract* approach, as well as, the inference rules defined on the logic, used to verify/prove properties and behavior of programs.

2.2 Design by Contract

The *Design-by-Contract* [22] is an approach for software development that assures high reliability by guaranteeing that the components of a system will run according its expectations. This technique, based on Hoare Logic (see Section 2.3), allows the specification of program required behaviour as contracts, boolean expressions that specifies the mandatory state before, during and after a program execution. This contracts are noted as assertions checked at systems runtime that validate the program correction if were all True, this means that when one assertion evaluates to False the software can be broken.

For initial state specification it is normally used Pre-Conditions defining a condition that the programmer caller must satisfy and for the final state Post-Conditions defining a condition that must be satisfied by the program execution. For instance, to maintain the consistency during program execution it is introduced a contract as Invariant, a condition that must be always True to validate the consistency of, for example, loops or conditional programs.

Design-by-Contract was firstly developed by Bertrand Meyer as design process associated with his Eiffel Object Oriented programming language, however, now is a valuable design technique for many programming languages, including Ada with SPARK.

2.3 Hoare Logic

C.A.R Hoare defined, in [23], a logic system with axiomatic and inference rules to reason about the correctness of a program. At the beginning the Hoare Logic only worked to prove the partial correctness of programs. However, was introduced new inference rules to prove the full correctness. The Hoare Logic is used to prove that the program behaviour agrees with the specifications, such as pre-conditions and post-conditions that are assertions defined on first order logic.

The Hoare logic fundamental semantic is called Hoare Triple. An Hoare Triple describes a contract between a program P and his caller. The caller had to satisfy a pre-condition A before the

execution of program P and P had to satisfy a post-condition B after. The Hoare Triple has the following form:

$$\{A\}\, P\, \{B\} \quad (2.1)$$

In 2.2, we have a trivial example of Hoare Triple on practice, where the pre-condition is $x = 3$, the program is $x := x + 10$ and the post-condition is $x > 3$. This example is totally correct, because if $x = 3$ adding 10 clearly implies that $x > 3$.

$$\{x = 3\}\, x := x + 10\, \{x > 0\} \quad (2.2)$$

Hoare defined the following inference rules to prove the validation of an Hoare Triple:

- **Conditional**: P execution or Q execution have to satisfy B. The choice between P and Q depends on E value, that is, if E is true ($A \wedge E$) implies P execution else if E is false ($A \wedge \neg E$) Q is executed;

$$\frac{\{A \wedge E\}\, P\, \{B\} \quad \{A \wedge \neg E\}\, Q\, \{B\}}{\{A\}\, \textbf{if } E \textbf{ then } P \textbf{ else } Q\, \{B\}} \quad (2.3)$$

- **Iteration**:

  - *Partial Correctness*: An loop invariant condition, I, had to be true at all iterations of the loop, that is, the loop starts on a state that satisfies I and ends on a state that satisfies I. If the loop ends the control condition now is false that implies the post-condition $\{I \wedge \neg E\}$.

$$\frac{\{I \wedge E\}\, P\, \{I\}}{\{I\}\, \textbf{while } E \textbf{ do } P\, \{I \wedge \neg E\}} \quad (2.4)$$

  - *Full Correctness*: The previous rule only prove the partial correctness, because there is no guarantee that the loop ends. Thus, the variant loop condition V ensures that the loop ends. The $W(<, P)$ it's a mapping of all the possible states and a defined subset of this states that is smaller at each iteration.

$$\frac{W(<, P) \quad \{I \wedge E \wedge V = n\}\, P\, \{I \wedge V < n\}}{\{I\}\, \textbf{while } E \textbf{ do } P\, \{I \wedge \neg E\}} \quad (2.5)$$

- **Composition**: Considering a program with a sequence of statements P and Q, the execution of the first, P, derives a post-condition B that will be the pre-condition of the second, Q.

$$\frac{\{A\}\, P\, \{B\} \quad \{B\}\, Q\, \{C\}}{\{A\}\, P; Q\, \{C\}} \quad (2.6)$$

- **Deduction**: New theorems are derived by other proved theorems. Thereby, if a program ends on a state that satisfies B so satisfies all assertions that B implies ($B \rightarrow B'$) and if starts on state that satisfies A satisfies any assertion that implie A ($A' \rightarrow A$).

$$\frac{A' \rightarrow A \quad \{A\}\, P\, \{B\} \quad B \rightarrow B'}{\{A'\}\, P\, \{B'\}} \quad (2.7)$$

2.4 Conclusions

The chapter introduced the formal methods thematic, important to understand the next chapter 3 where, SPARK verification toolset is presented. SPARK performs formal code verification that is a kind of static analysis where the specifications follow the *Design-by-Contract* approach based on Hoare logic.

Chapter 3

Tools

3.1 Introduction

SPARK analyses the information flow and proves key properties of the code that are annotated amongst Ada code with the SPARK contracts. This chapter starts with a short introduction of Ada programming language, followed by a presentation of SPARK, illustrating all the features of the toolset that can be applied to the Ada programs.

3.2 Ada Programming Language

Ada programming language is a strong typing language that performs dynamic verification for run-time errors as buffer overflows. The main target is real-time embedded software where high levels of safety, security and reliability are required such as Aerospace, Defence, Aviation, Railway and Space.

The most recent release, Ada 2012, added new features for programs behaviour specification such as contracts and type invariants that are transformed in assertions to be checked during run-time. Unlike the SPARK that checks the annotated contracts statically, in Ada 2012 the specifications can be checked dynamically during the process of unit testing[2].

3.3 SPARK

SPARK, developed in a collaboration between AdaCore and Altran, is a programming language targeted to static formal verification of Ada code. It is an Ada subset that applies restrictions to the Ada programs to be sure that his implementation is in conformance with the program specification and will behave as expected. This restrictions are annotations added to the code that follow the *Design-by-Contract* approach and describe the specifications.

The most recent version is SPARK 2014 introduced the GNATProve toolset to statically perform flow analysis and formal code verification. GNATProve toolset uses Why3 platform as intermediate to perform deductive verification.

SPARK has been used during the years in on-board aircraft systems, control systems, cryptographic systems and railway systems [24][2][25] as represented in Chapter 4.

3.3.1 Flow Analysis

Firstly, the flow analysis detects many types of errors or violations that are not supported by SPARK. The errors or violations are the followed:

- **Aliasing of names** is forbidden by SPARK because it turns the verification more complicated and the results can be different as expected;
- **Access Types** are not valid too because it can possible create aliasing problems;
- **Side effect functions** are functions where global variables or passed by parameter variables are updated. In Ada a function is only defined to compute and return a value not to update other object, so side effects is forbidden to avoid unpredictable behaviour.

Finally, the main concerns of flow analysis is data. It checks if functions parameters or global variables are correctly defined, initialised or attributed. Flow analysis also checks the information flow during the programs execution and the use of global variables with additional contracts, Depends and Global, respectively. Only *in out* or *out* parameters or updated global variables dependencies can be specified.

3.3.1.1 Depends Contracts

Depends Contracts can be assigned to each subprogram specifying dependencies between inputs and the final outputs. In Listing 3.1, we have a procedure specification where the Y value will be added to X value, so the contract define the dependencies of X variable. Detailed, the X final value depends on its initial value and Y value.

```
1 procedure ADD_Y_TO_X(X : in out Integer; Y : in Integer) with
2     Depends => (X => (X,Y))
3 ;
```

Listing 3.1: Depends Contract example.

3.3.1.2 Global Contracts

Global Contracts as Depends can be assigned to each subprogram specifying the use of global variables and if they are accessed or updated. In Listing 3.2 we have an example with the four possible annotations. For variables noted as *Input* means that their values are only for read, *Output* means that their values must be modified, *In_Out* means that their initial values are for read and must be modified and *Proof_In* means that theirs values is only for code proof.

```
1 procedure P with
2     Global => (Input    => (A, B, C),
3                Output   => (L, M, N),
4                In_Out   => (X, Y, Z),
5                Proof_In => (I, J, K),)
6 ;
```

Listing 3.2: Global Contract example.

3.3.2 Formal Code Verification

Formal Code Verification can be resumed in one word, ”proof”. Proof that the code is free of run-time errors and proof that code implementation has the expected behaviour, in conformance with the requirements.

3.3.2.1 Absence of Run-Time Errors Proof

Prove the absence of Run-Time Errors is the way to prove the program's integrity. Run-Time errors cannot be detectable on compilation phase, however with SPARK are detected statically with GNATProve. The possible errors detected by SPARK are the followed:

- Divisions by zero;
- Range violations;
- Overflows;

Part of this properties are only proved with the addition of SPARK aspects such, properties added to Pre-Conditions.

3.3.2.2 Functional Properties Proof

Prove functional properties can be seen as proving correctness of programs, that is, checking if the program implementation complies with the requirements. Here, we are now addressing the *Design-by-Contract* approach. Where a Pre-Condition and a Post-Condition (3.3.2.3) will establish a relation between the initial and final state of the program.

Sometimes, the functional contracts are not automatically proved and is necessary additional annotations (3.3.2.4, 3.3.2.5) to guarantee that some properties are satisfied during program execution. Using Ghost Code (3.3.2.6) is also helpful for this purpose.

3.3.2.3 Functional Contracts

As said above 3.3.2.2, the Pre-Conditions and Post-Conditions establish a relation between the program initial and final status giving the guarantee that the code has the correct behaviour, as required. A Pre-Condition is a property that has to be verified when the program is called and a Post-Condition is a property that has to be verified at the end of program execution.

In Listing 3.3 we have an example of the use of functional contracts in SPARK, a procedure that increment X value. The Pre-Condition means that X initial value has to be lower then the last integer ensuring that there is no "Integer Overflow" and the Post-Condition means X final value has to be bigger then X'Old (initial X) ensuring the program goal.

```
1 procedure Increment_X(X : in out Integer) with
2     Pre  => X < Integer'Last,
3     Post => X > X'Old
4 is begin
5     X := X + 1;
6 end Increment_X
```

Listing 3.3: Functional Contracts example.

SPARK 2014 new feature is the Contract Cases, that allows the specification of the required condition for each distinct case. In Listing 3.4 we have a procedure that increment the X initial value if is lower then 10. The Contract means that if initial X is lower then 10, X has to be incremented, else X value is maintained.

```
procedure Add_One(X : in out Integer) with
    Contract_Cases => (X < 10  => X = X'Old + 1,
                       X >= 10 => X = X'Old)
is begin
    if X < 10 then
        X := X + 1;
    end if;
end Add_One;
```

Listing 3.4: Contract Cases example.

3.3.2.4 Assertions

In a conditional program, for instance, some properties are not equal in all stages thus, to help the functional contracts verification the code can be annotated with the pragma Assert to ensure that the properties for the Post-Condition verification are verified during execution. The condition defined on the pragma Assert is always verified. In Listing 3.5, a trivial example of a pragma Assert definition in SPARK.

```
procedure Assert(X : in out Integer)
is begin
    X := 1;
    pragma Assert(X = 1);
end Assert;
```

Listing 3.5: Pragma Assert example.

There is the Pragma Assume, where it is possible to define a condition that SPARK has to assume verified to prove other annotations. In Listing 3.6, we have the same trivial example where the pragma Assume is replacing the statement that attributes 1 to X.

```
procedure Assume(X : in out Integer)
is begin
    pragma Assume(X = 1);
    pragma Assert(X = 1);
end Assume;
```

Listing 3.6: Pragma Assume example.

3.3.2.5 Loop Invariant and Loop Variant

Loop Invariant has the same function as assertions, ensuring properties during the loops execution. The loop implementation is assumed correct if the property defined on the Loop Invariant is valid at every loop iterations.

According with what is defined on Hoare Logic, Section 2.3, the Loop Invariant only proves the loop partial correctness thus, it is necessary a Loop Variant condition to prove the loop termination. The Loop Variant is a value that has to decrease at each iteration.

In Listing 3.7, we have a procedure that will decrease the X value at each iteration until X value be 0. The Loop Invariant means that at every iteration the X value has to be bigger then 0. If is 0 could not entry on the loop and, the Loop Variant prove that at each iteration the X value is decremented until 0 assuring the loop termination.

```
procedure Decrease_X(X : in out Integer)
is begin
    while X > 0 loop
        pragma Loop_Invariant(X >= 0);
        pragma Loop_Variant(Decreases => X);
        X := X - 1;
    end loop;
end Decrease_X;
```

Listing 3.7: Loop Invariant and Loop Variant example.

3.3.2.6 Ghost Code

Sometimes the variables and functions that are useful for the implementation are not sufficient for the contracts specification and it is possible to use additional variables and functions that will only be used for code verification [26].

Ghost Code will not affect the program behaviour and only is used for code verification. It is possible to define ghost variables, types, procedures, functions and packages. Ghost variables are normally used to track the program state, procedures to update the ghost variables and functions are used to define properties used on contracts.

In Listing 3.8 the variable *Prop* is used to track the state of procedure *P* that do some computations on X and Y and execute *Compute_Prop* a ghost procedure that computes *Prop* result. The ghost function *Is_Prop_Valid* is used on the Pre-Conditions and Post-Conditions to check if *Prop* computation was correct. It is a trivial example just to illustrate how ghost code can be used.

```
Prop : Integer := 0 with Ghost;

function Is_Prop_Valid return Boolean is (Res in Integer'Range) with Ghost;

procedure Compute_Prop(X, Y : in Integer)
with Ghost
is begin
    Prop := Y * X;
end Compute_Prop;

procedure P(X, Y : in out Integer)
with
    Pre  => Is_Prop_Valid,
    Post => Is_Prop_Valid
is begin
```

```
16      -- some computations of X and Y
17      Compute_Prop(X,Y);
18  end P;
```

Listing 3.8: Ghost Code example.

3.3.3 SPARK Levels

Before starting the SPARK application on a project, one should define the scope for the context, or what is possible to reach, with this application. The level of analysis may range from a simple flow analysis to the prove of complex properties [27]. The SPARK use can be divided in 5 levels of assurance:

- **STONE -** Valid SPARK code. It is the first phase of flow analysis, where features not supported by SPARK are detected;
- **BRONZE -** Initialization and correct data flow. Analysis of flow contracts (Global and Depends);
- **SILVER -** Absence of run-time errors;
- **GOLD -** Proof of critical properties;
- **PLATINUM -** Proof of full functional properties;

In [1], according with last Altran UK industrial projects results, they summarize the application of different levels of assurance compared with the most common software integrity scales (Design Assurance Level (DAL) defined in DO-178B and Software Integrity Level (SIL) defined in IEC 61508), see Figure 3.1.

Software Integrity Level		SPARK Verification Objective			
DAL	SIL	Bronze	Silver	Gold	Platinum
A	4				
B	3				
C	2				
D	1				
E	0				

Figure 3.1: Technical Planning Guidelines for the Application of SPARK.[1]

With Silver as default level to Platinum level (Black region) they achieved the highest levels of integrity. On the medium levels of integrity (Grey region) Silver still the default level, although, in this case the most advanced achieved level is Gold. And final, the lowest levels of integrity (White region) are achieved with Bronze as default and the most advanced level is Silver.

3.3.4 GNATProve

In SPARK 2014, was introduced the new toolset for formal verification that performs the Flow Analysis and Formal Code Verification. GNATProve [28] is a toolset for contract-based formal verification of Ada subprograms based on GNAT compiler in GCC developed on Hi-lite project [29] to

perform the same functionalities as SPARK 2005 verification toolset. It generate the Verification Condition (VC)'s[1] to be proven by automatic provers.

GNATProve, auxiliary, uses the Why3 platform [17] to perform the code verification. As we can see on Figure 3.2. The SPARK code is translated to WhyML, that is the specification language of Why3 [17], by Gnat2Why [28] tool, and Why3 platform transforms the WhyML contracts in VC's, defined in first order logic. Finally, these VC's will be proven by Satisfiability Modulo Theories (SMT) provers such as Alt-Ergo[30], CVC4[31] and Z3[32].

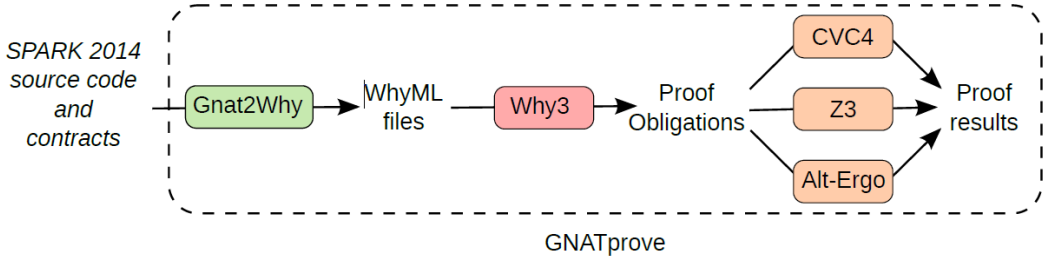

Figure 3.2: Dedutive Verification with GNATProve.[2]

In [28], [24] and [2] we have a more detailed explanation about how GNATProve does the translation for WhyML and the Why3 deductive verification is performed.

3.3.4.1 GNATProve Report

Using the GNAT Programming Studio, after the SPARK code analysis, two types of reports are produced by GNATProve. One shows on the Integrated Development Environment (IDE) all proved and unproved checks as shown in Figure 3.3 and the other is a file generated containing a textual summary of the analysis results as shown in Figure 3.4. Note that this two examples refers to Flow Analysis and Formal Code Verification jointly, although it can be performed in separation.

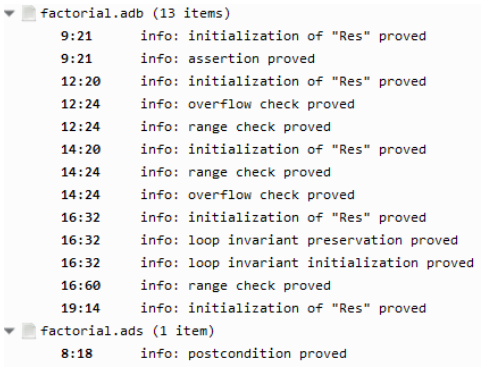

Figure 3.3: IDE Report example.

The IDE report, Figure 3.3, only show proved checks however GNATProve, when exists unproved checks, generate potential counterexamples that gives values for the programs variables, showing a particular case where the annotation might not be valid [24].

[1] Logical formula whose validity must be proved to ensure that a corresponding assertion is verified.

```
Summary of SPARK analysis
=========================

-----------------------------------------------------------------------------------------------------------
SPARK Analysis results        Total       Flow   Interval   CodePeer                Provers   Justified   Unproved
-----------------------------------------------------------------------------------------------------------
Data Dependencies                 .          .          .          .                      .           .          .
Flow Dependencies                 .          .          .          .                      .           .          .
Initialization                    5          5          .          .                      .           .          .
Non-Aliasing                      .          .          .          .                      .           .          .
Run-time Checks                   5          .          .          .   5 (CVC4 89%, Z3 11%)           .          .
Assertions                        3          .          .          .              3 (CVC4)           .          .
Functional Contracts              1          .          .          .   1 (CVC4 75%, Z3 25%)           .          .
LSP Verification                  .          .          .          .                      .           .          .
-----------------------------------------------------------------------------------------------------------
Total                            14    5 (36%)          .          .              9 (64%)           .          .
```

Figure 3.4: File Report example.

The rows of file report, Figure 3.4, shows the types of verifications that are performed and the columns shows, for each type, as follow:

- **Total:** number of checks performed;
- **Flow:** number of checks proved by flow analysis;
- **Interval:** overflows and range checks proved by type ranges;
- **Provers:** number of checks proved by automatic or manual provers;
- **Justified:** number of check proved by the user;
- **Unproved:** number of unproved checks;

3.4 Conclusions

This chapter presented an overview of Ada programming language, a detailed presentation of the Ada formal code verification tool - SPARK, and all static analysis that it is capable to perform.

The ExoMars software was implemented with an old version of Ada language, Ada 2005, but it was not a problem because this old version is supported by the most recent, Ada 2012, and directly with SPARK 2014 Discovery without big changes. Therefore, SPARK 2014 Discovery was the version used to comply the target goals, including all features presented in this chapter and GNATProve to perform the analysis.

Chapter 4

Formal Verification with SPARK : Related Work

4.1 Introduction

It is a fact that a failure on a Safety Mission-Critical software can result in significant damage to the organizations or even to the population in general. A simple wrong data type conversion can result in mission loss. Looking to the past, there are a set of cases that testify the assertions above, for instance, the Ariane 5 explosion and the Schiaparelli (ExoMars mission) crash landing on Mars surface.

The Ariane 5 explosion results due to a rocket velocity (64-bit float) dependent variable declared as 16-bit float that exceed the bounds with the fast rocket speed increase, that is, this value needed a 64-bit float type. The failure report is available in [33]. Other example, a payload, the Schiaparelli, with the same characteristics as the Rover that will be launched on Mars, was part of ExoMars TGO in 2016 to approve the Entry, Descent Landing (EDL) system. A failure on the distance to the Mars surface measure implied a late parachute activation and the payload crash to the surface at a great speed. The failure report is available on [34]. This two failures and many others are good examples that show the needed of automated techniques as SPARK to detect aspects that often are undetectable by software tests.

The SPARK adoption has been increased in various domains essentially on safety critical systems. This chapter presents a general overview of the work done until the date using SPARK formal verification. The following sections are organised as follows:

- **Section 4.2:** An overview of the work of under graduated students with SPARK for space exploration;
- **Section 4.3:** An study case for SPARK application on the development of generic flight software;
- **Section 4.4:** As SPARK for space is a reduced state of the art thus, in this section an overview of the work besides the space exploration.

4.2 Vermont Lunar CubeSat

Vermont Technical College joined Educational Launch of Nano-satellites (ELaNa) IV mission of NASA with other 12 university researchers. They were selected to implement a CubeSat to be launched on earth orbit at 500 Km of altitude. A CubeSat (or Nano-satellite) is a spacecraft of small dimensions similar to a cube with a volume of 10cm^3 and 1.33Kg. Vermont Lunar CubeSat [35, 36, 37, 38] was the only university CubeSat that operates successfully all mission, about 30 months.

In the ELaNa mission the goal for Vermont Lunar CubeSat was to test the navigation software that will be used on the next mission to the moon, the Lunar IceCube 4.2.1, for ice and water measure in the lunar surface. The navigation system used Goddard Enhanced Onboard Naviagtion System (GEONS), developed by NASA Goddard Space Flight Center in C, re-written in SPARK/Ada. GEONS receives Global Positioning System (GPS) signals and images of a star tracking camera, process the GPS signal and compute the orbit propagation. Only 25% of GEONS were translated for the ELaNa IV mission however, many run-time were still found.

Although C is the most common language used for CubeSats development, Vermont Lunar CubeSat was totally written in SPARK. The system was based on a Texas Instruments MSP430F2618 16-bit micro-controller provided on the CubeSat Kit. There is no Ada compiler available for this micro-controller, therefore, it was performed a well-known approach on a later project, the Artic Sea Ice Buoy (See Section 4.2.2), where the same CubeSat Kit was used and the Ada code verified by SPARK was translated for C.

The Vermont Lunar CubeSat development final results were:

- **SPARK Code:**
  - 5991 lines of code;
  - 4095 lines of comments where 2843 are SPARK annotations [1];
  - Generated 4542 Verification Conditions, 98% automatically proved;
- **C Code:**
  - 2239 lines, blanks lines included;

The mission was a success and the next path was continue the GEONS translation for the development of an operating system designed for flight control of CubeSats, the CubedOS [39], used as support on Lunar IceCube 4.2.1 development. CubedOS is a modular software platform for CubeSat missions that makes the CubeSats flight software development easier. CubedOS can operates as an operational environment or as tool and useful functions library. All CubedOS modules were proved free of run-time errors and some critical sections was verified.

4.2.1 Lunar IceCube

Lunar IceCube[40] is an 6 unit spacecraft that will be launched on National Aeronautics and Space Administration (NASA)'s Space Launch System EM-1. The project is a collaboration between Vermont Technical College, Morehead State University, NASA's Jet Propulsion Laboratory, NASA's Deep Space Network (DSN), NASA's Goddard Space Flight Center and Busek Inc..

The software will control the de-tumbling by low power, that runs out past 30 minutes after deployment. The spacecraft will be deployed on a sun centric orbit so the software will control the solar arrays to be oriented to the sun before the power runs out. The software also will be in constantly contact with DSN to receive and process commands coming from the earth that will guide the spacecraft up until the lunar orbit. Once in lunar orbit the flight software will control the science mission and mapping the water vapour and ice, autonomously.

[1] In SPARK 2005 the annotation were specified as code comments.

For software verification the main goal is to prove that the software is free of run-time errors and after that, try to prove the correctness of the most critical software properties. Additionally, some unit tests will be performed to the produced code. In Lunar IceCube it has been used the most recent release of SPARK, the SPARK 2014 that has a development environment more powerful than SPARK 2005. For example, the new feature of SPARK 2014 is that it supports the Ravenscar Profile, useful for a periodic task that pools a database of pending commands.

4.2.2 Artic Sea Ice Buoy

The Artic Sea Ice Buoy project [3] was a collaborative work of Vermont Technical College and University of Vermont. It is a buoy that collect environmental data and send to Vermont with GPS and sensors for temperature and wind velocity and direction measurements. The CubeSat kit used in the Vermont Lunar CubeSat was firstly used in this project. This kit has a Texas Instruments MSP430F2618 16-bit micro-controller working has CPU, that does not include any Ada code compiler. Thereby, to comply this issue was used the AdaMagic tool that converts the Ada code in C code and after with Rowley Associates CrossWorks C compiler for MSP430 compile the translated code and produces the binary. The workflow is represented on Figure 4.1, where the square boxes are code and the rounded boxes are the tools used to process the code.

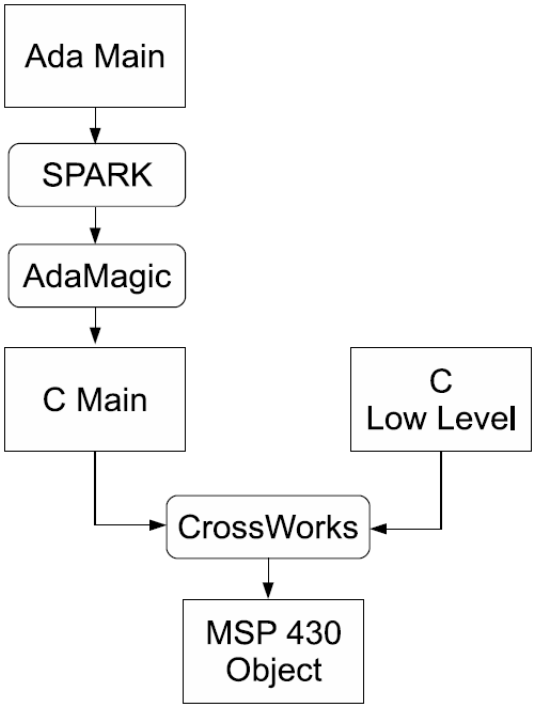

Figure 4.1: Artic Sea Ice Buoy workflow [3].

It is important to note that, as is understandable on the Figure 4.1, before the Ada code be translated by AdaMagic, was performed the software verification with SPARK to prove that software is free of run-time errors.

4.3 Automated Transfer Vehicle Solar Generation Subsystem

Formal Model Driven Engineering for Space On Board Software [41, 10, 42] ,a study case developed by Astrium Space Trasnportation (AST) for the use of formal methods on space software

development. In this study was developed the Solar Generation Subsystem (SGS) of Automated Transfer Vehicle (ATV), a spacecraft that supports the International Space Station developed by AST to ESA. The SGS controls the deployment and the rotation of ATV solar wings.

For automatic software architecture modelling of solar wings deployment based on the architecture defined with OMEGA SysML was used SCADE, and SPARK code was automatically generated by SCADE KCG Ada. In addition, a set of algorithms was written manually for better results gaining energy. With SPARK toolset was proved the absence of run-time errors in the generated SPARK code (97% of VC's proven automatically and 3% with a theorem prover).

To evaluate the Hi-Lite project [29] adequacy of formal verification on space software this tool was applied to the Flight Control software and Mission and Vehicle Management generated. In the Subsections 4.3.1 and 4.3.2 we have the results.

4.3.1 Flight Control

In this case, the analysed code computes float values received as inputs applying math operations to this values. This study only aims to prove the absence of run-time errors and the correct ranges of variables due to the difficult to define functional contracts.

In the table 4.1 we have the final results. The percentage of generated VC's, for each type of SPARK annotation, proved.

	Mathematical Library		**Numerical Algorithms**	
Features	Generated VC's	Proved (%)	Generated VC's	Proved (%)
assertion	-	-	42	100
division_check	3	100	9	100
index_check	-	-	2	2
overflow_check	9	100	95	100
postcondition	4	75	2	100
precondition	1	100	34	100
range_check	10	90	81	96
Total	27	92	265	98

Table 4.1: Flight Control results.

4.3.2 Mission and Vehicle Management

In this case, the analysed code is a generic On-Board Control Procedure that can be used in a launcher or in a spacecraft that follow all ECSS-E-ST-70-01C principles.

This study aims to verify the software tables, the consistency between Mission and Vehicle Management (MVM) and other functionalities such as the solar wings management, critical properties and the absence of run-time errors.

In the table 4.2 we have the final results. The percentage of generated VC's, for each type of SPARK annotation, proved.

	Generated VC's	Proved (%)
Time Management	3	100
Mathematical Library	137	97
Single Variables	268	100
List of Variables	252	100
Events	213	100
Expressions	1670	100
Parameters	31	93
Functional Unit	106	74
Automated Procedure	284	74
On Board Control Procedure	2454	95

Table 4.2: MVM results.

4.3.3 Final Results

In the final report [43] is represented a table, similar to table 4.3, with conclusions about the performance of SPARK during the study, where it is presented the evaluation for every specific requirements that SPARK shall fulfill.

In almost the cases the results was "Very Good", essentially flow analysis and the proof the absence of run-time errors. Relatively to the "Proof of functional properties" the evaluation was lower due to the complexity of the annotations needed to achieve this requirement.

Requirements	**Assesment**
Formalisation of the test cases	Very Good
Test cases coverage	Not implemented
Proof of absence of run-time errors	Very good
Proof functional properties	Good
Correct access to all global variables	Very Good
Absence of out of range values	Very good.
Internal consistency of software unit	Very good
Dead code detection	No
Correct numerical protection	Very Good
Correctness of a generic code in a specific context	Very good

Table 4.3: SPARK use evaluation.

4.4 Other Domains

The industrial use of formal methods in space domain is still low. Many organizations are still afraid of the mathematical complexity of this approach. The statistics including SPARK and its use on space domain is limited too. Thus, in this section, complementary to the illustrated cases in Section 4.2 and Section 4.3, other cases of other domains(aerospace or security) are shown.

The old Praxis High Integrity Systems, now Altran UK, has a big experience in the past with SPARK. The best outcomes are described by [44, 25, 45, 46]. In the next subsections are described this projects and more recent projects from other organisations.

4.4.1 SHOLIS

The Ship Helicopter Operating Limits Information System (SHOLIS) [4] is an information system that advice the helicopters for safe operations on naval vessels. SHOLIS is used on UK Royal Navy and Royal Fleet Auxiliary vessel. Its software applications development was in charge of Altran UK subcontracted by UK Ministry of Defence.

The system has a database of the allowed limits for performing a specific operation for a specific type of helicopter and is constantly making comparisons between sensors information and the limits defined on the database. It was developed under UK Interim Defence Standards 00-55 and 00-56, which advocate the use of formal methods on systems that are Safety Mission-Critical. It was the first project developed to the full degree rigour required by these standards.

Following the standards mentioned, the development process was divided in Requirements, Software Requirements Specification (SRS), Software Design specification (SDS), coding and testing. For the SRS was used Z notations [47] and natural language (English), for the SDS was used SPARK and Z notations and natural language and for coding was essentially used the SPARK programming language, the version SPARK 83.

The proof activities on the SHOLIS project were divided in two different proofs. One, the Z proof, at the SRS and SDS levels, used the CADiZ tool [48] as assistance to the proofs activities for schema expansion. Other, the SPARK proof, at the code level, used all the tools performed by the SPARK toolset (Examiner, Simplifier and Proof Checker) to support the proof activities. It was performed flow analysis for all SHOLIS code, proof of key properties using Pre-Conditions, Post-Conditions and other contract aspects and proof the absence of run-time errors.

The resulting metrics of all proof activities during the SHOLIS project were:

- On the Z proof work, were executed 150 proofs, which 130 were at the SRS level and the other 20 were at the SDS level;

- On the SPARK proof work, were generated 9000 VC's. 3100 proofs of key properties and 5900 from run-time checks to the proof of absence of run-time errors;

Other results about the project effort, that are resumed on Table 4.4, show that the Z was the most efficient phase for bugs finding and the next one was the System Validation Test and Code proof with SPARK was more efficient than Unit Testing.

Phase	Faults found (%)	Effort (%)
Specification	3.25	5
Z proof	16	2.5
High-level design	1.5	2
Detailed design, code and informal test	26.25	17
Unit Test	15.75	25
Integration Test	1.25	1
Code proof	5.25	4.5
System validation test	21.5	9.5
Acceptance test	1.25	1.5
Other	8	32

Table 4.4: Faults Found and Effort Spent During Phases of the Project [4]

In spite of the code proof with SPARK have not been the most efficient phase, it was important to prove that the code was free of run-time erros and to prove some functional properties. The flow analyis was also important because detected several vulnerabilities of Ada language (e.g. aliasing cases).

4.4.2 MULTOS CA

Multi-Application Operating System (MULTOS)[49] is a smart-card Operating System (OS) that allows the operation of multiple applications running on a single MULTOS Carrier Device (MCD) . The MULTOS Certification Authority (CA) [5] sign the digital certificates included in the data used to enable the smart-card and the applications, preventing frauds.

The system was designed to meet the security constraints on level E6 of UK ITSEC standards. It is a distributed system with one computer as the user-interface and a set of isolated industrial computers that performs all critical security functions such as sign certificates, files encryption and cryptographic keys generation.

The implementation was divided in many different languages with SPARK useful to implement all systems security functions. On table 4.5, we have the representation of the used languages and its utilities.

SPARK	30%	Security Kernel
Ada 95	30%	Concurrency, tasks, processes, database interfaces, bindings to ODBC and Win32
C++	30%	GUI
C	5%	Device drivers and SHA1
SQL	5%	Database

Table 4.5: The main functions performed by each programming language used [5]

The SPARK data-flow errors detection and the information flow analysis had promoted a high cohesion and loose coupling of components, and made the process easier every time the requirements have changed.

4.4.3 LOCKHEED C130J

LOCKHEED C130J [50] , normally known as ”Hercules”, is a long line military and commercial transport aircraft. In the second version implementation [51] , LOCKHEED Aeronautical Systems Company (LASC) and Altran UK prioritise the Mission Computer implementation of the new avionic system.

LASC specified the software requirements in a tabular form using Consortium Requirements Engineering (CoRE) [52], a formal requirements modelling method that specify mathematically the input-output relationships. There are a more detailed illustration in [53].

For the implementation process, Altran UK derives the SPARK annotations from the CoRE and from the design documents. In other words, the specifications in CoRE were translated to Post-Conditions in the SPARK programs, there are examples of this procedure available in [51]. The SPARK Examiner, SPADE Automatic Simplifier and SPADE Proof Checker were used to prove that the code meets its specifications.

In [51], concluded that SPARK ”was crucial in binding the code to the initial requirements”, with an 80% saving in the expected budget allocated for system testing.

4.4.4 Tokeneer ID Station

The Tokeneer system is a system developed by National Security Agency (NSA) [54] as a research project to investigate biometric aspects applied to access controls. It is a system that uses biometric information of users (e.g. fingerprint) saved on a smart-card to control the people access to a specific room. The Tokeneer ID Station (TIS) is the component of Tokeneer system in charge of processing and verifying the user biometric information and decide if he had permission to unlock the door or not.

For this project [55, 56, 57], the NSA contacted the Altran UK to re-develop TIS applying his *Correction-by-Construction* approach. The goal was to demonstrate that is possible develop a high quality, low defect system in a cost-effective way in conformance with the requirements defined on CC - EAL5.

To following the CC - EAL5 requirements was defined 6 different stages for the development process. Initially, for requirements analysis, the Altran UK REVEAL approach was applied ensuring the use of the right tools at the right time and the documentations of the key information and decisions. The intended behaviour for TIS was rigorously and formally specified using the Z specification language annotations and texts in English natural language. Next, the INFORMED Design process was important to provide information about the systems architecture to pass from the Design to the implementation without misunderstandings about the system. Thus, in the implementation, the code was written in Ada/SPARK and after was verified by the SPARK Examiner toolset. And finally, the system was tested just to demonstrate that the systems had the correct behaviour as specified.

In the table 4.6, we have the metrics that resume the project development effort.

LOC	9939
Total Effort (days)	260
Productivity(LOC/per day, overall)	38
Productivity(LOC/per day, code)	203
Defects after Delivery	4

Table 4.6: TIS development effort.

In the end of the project, the CC - EAL5 levels of assurance were achieved and they assumed that the CC - EAL7 can be achieved too when his *Correctness-by-Construction* process is followed.

4.4.5 iFACTS

Interim Future Area Control Tools Support (iFACTS), developed by Altran UK contracted by National Air Traffic Services (NATS) [58], is an air-traffic control system used in NATS London area control operation.
Not much information about the project and his SPARK use is available. Probably it is because the organisations confidential criteria but, it is known [25] that the functional specification was formally expressed in Z and SPARK was directly used for the proof on type-safety excluding the system functional correctness due to his rigorous requirements.
Were about 529k lines of code where 250k are logic code and 74k are SPARK contracts, with 152927 VC's generated (98.76% proved automatically).

4.4.6 ANSSI - WooKey

Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI) [59], is designing and implementing an USB key platform that provides security against vulnerabilities on the hardware, architecture, protocol and software stack. WooKey Platform [60] is a USB thumb drive designed for data encryption and protection with a set of security defences.
The project is a work in progress that uses Ada with SPARK as safe languages to write the kernel critical parts and the components belonging to the Trusted Computing Base, that are directly connected with the platform security and exposed to untrusted applications and user inputs.
SPARK formal verification has been helpful detecting errors that could occur during run-time execution, for instance, out-of-bound array accesses, integer overflows and dangling pointers.

4.5 Conclusion

The use of formal methods is always an optional choice when addressing the verification and validation of systems that are Safety Mission-Critical. That is mainly due to the complexity of the annotations needed to prove that the system comply with all rigorous requirements that these type of systems entail. Nonetheless, it is changing, and the use of formal methods is increasing, including SPARK. For instance, the first use of SPARK in space domain was a success, an academic Cubesat 4.2 that remained on the lunar orbit during a mission of 30 months with the flight software verified by SPARK that path to the implementation of generic flight software that can be used by any payload. Still in the space domain, a study case 4.3, tested the SPARK use on spacial software following the principles of a rigorous standard, proved how it is good in aspects such flow analysis and proof the absence of run-time errors and functional properties.

Altran UK gaves a good contribution to the state of the art related to the use of SPARK, developing many of the projects described above (See Subsections 4.4.1 , 4.4.2 , 4.4.3 , 4.4.4 and 4.4.5). In [44], we have have a resume table with metrics that shows how the median low defect rates in that projects was very good. For example, in TIS, Subsection 4.4.4, the defect rate is 0% of defects per 1000 lines of code produced and MULTOS CA , Subsection 4.4.2, had a 0.04% rate.

Recently, many different projects of various domains are adopting the SPARK formal verification. An example of a work in progress, WooKey 4.4.6, that is proving the correctness of critical functions to assures the security on USB platforms. And many other that we can follow the news in [61].

As this chapter states, SPARK usage over the years has been making a relevant contribution for systems development where safety, security and reliability are mandatory concerns. The final results of this investigation, see Subsection 7.3, was a good contribution for the formal verification adoption, especially with SPARK, on Safety Mission-Critical domains as space exploration and was an incentive for the adoption by the company where formal verification is not yet considered the most profitable solution.

Chapter 5

ExoMars TGO

5.1 Introduction

The ExoMars TGO is part of an ESA mission with the goal of investigate the martian atmosphere measuring, essentially, the levels of methane gas. The Critical Software, SA. was integrated on the development of CSW components applications, where was applied the formal verification technique with SPARK Toolset on this research work. In this chapter will be described the ExoMars TGO mission and his avionics system and on-board software architectures. Describing, at the end, in more detail the software component chosen for this research work;

5.2 Mission Overview

The ExoMars missions are the first of Aurora programme [62], aiming the environment investigation of Mars and demonstrating new technologies for a future Mars return mission in the 2020's. There are two missions on ExoMars programme:

- An Orbiter (TGO) plus an Entry, Descent and Landing Demonstrator (*Schiaparelli* [63]). Launched on 2016;
- A Rover, to be launched on 2020;

Technologically, the main goal of ExoMars missions is develop and qualify:

- EDL of a payload on the Martian surface;
- Surface mobility using a rover;
- Sample acquisition by accessing sub-surface;
- Sample acquisition, preparation, distribution and analysis;

The first mission was the TGO plus an Entry, Descent Module (EDM). The purpose of this mission is searching for methane and other atmospheric gases that could be evidence of biological and geological activities and testing technologies for the next ESA's mission to Mars.

A scientific payload responsible for the detection and characterisation of trace gases in the Martian atmosphere is transported by TGO. The instruments on-board of TGO will be deployed, on a science orbit altitude about 400-Km, to detect a wide range of atmospheric trace gases (methane, water vapour, nitrogen dioxide, acetylene) with a refined accuracy (three orders of magnitude compared to the oldest measurements). This measurements gives information about location and gases sources that could be the primary sites for landing on the next missions.
Further, the TGO transported the EDM from Earth to Mars deployed it for entering on the Martian atmosphere and landing on the planet surface, in a predefined area. After deployment, the TGO mission is monitoring the Ultra Hight Frequency (UHF) transmission from the EDM till

the landing on the Mars surface facilitating the real-time transmission of the important data measured by EDM to the ground station.

On the second mission, when the rover lands on Mars, the TGO will provide telecommunications services to the Rovers operation, acting as a data relay centre sending commands to the rover and downloading data to the ground station through the ESA space communications network.

5.3 TGO Avionics

The TGO Avionics is based on a central computer called Spacecraft Management Unit (SMU). On Figure 5.1 is represented the TGO Avionics architecture where the SMU communicates with the payload module, all TGO units and with the EDM.

The payload module is composed of Power Conditioning and Distribution Unit (PCDU), science instruments and UHF subsystem. The PCDU function is storing payload data and interfacing with the SMU through 1553 bus.

All software applications that belongs to the CSW are based on the SMU.

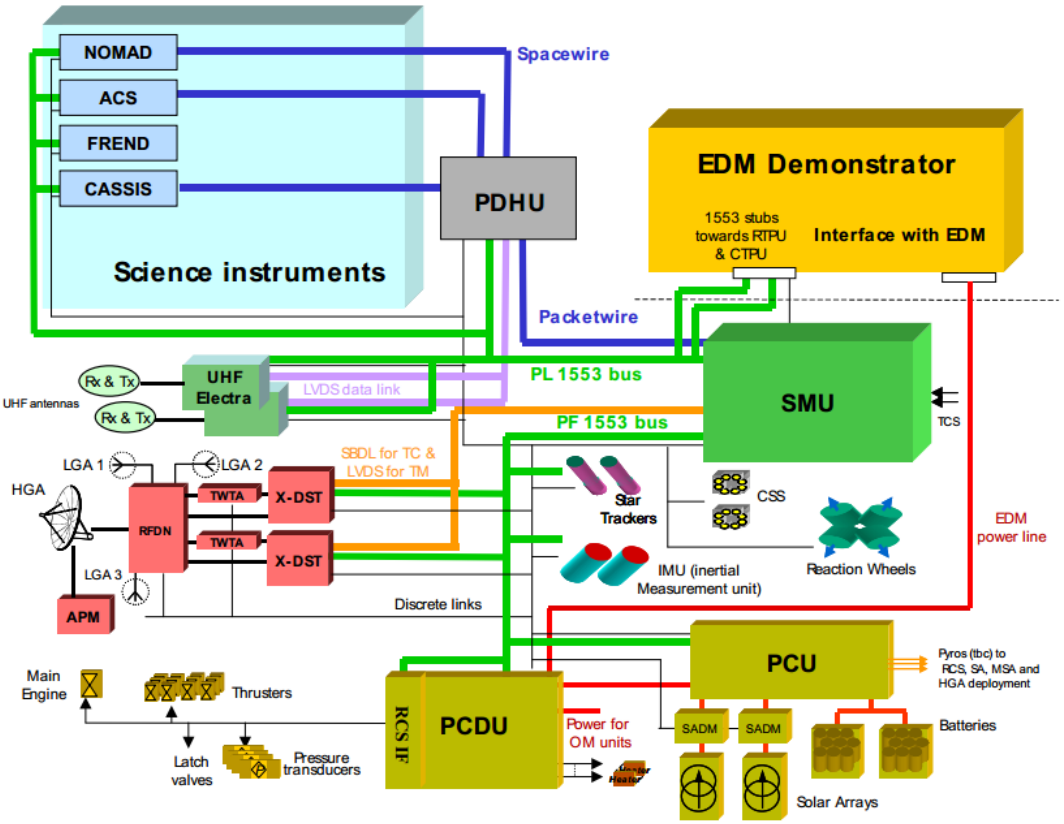

Figure 5.1: ExoMars TGO Avionics diagram.

Note: Figure 5.1 is from ExoMars TGO technical documentation.

5.4 ExoMars TGO CSW

The TGO on-board software relies on Thales Alenia Space - France (TAS-F) framework based on the reuse of software components common to several missions focusing the development on the components that are mission specific. It was already used on Sentinel-3 [64] which Critical Software, SA. was involved on the development too. The Figure 5.2 represents the architecture defined by the TAS-F framework adapted to the TGO mission.

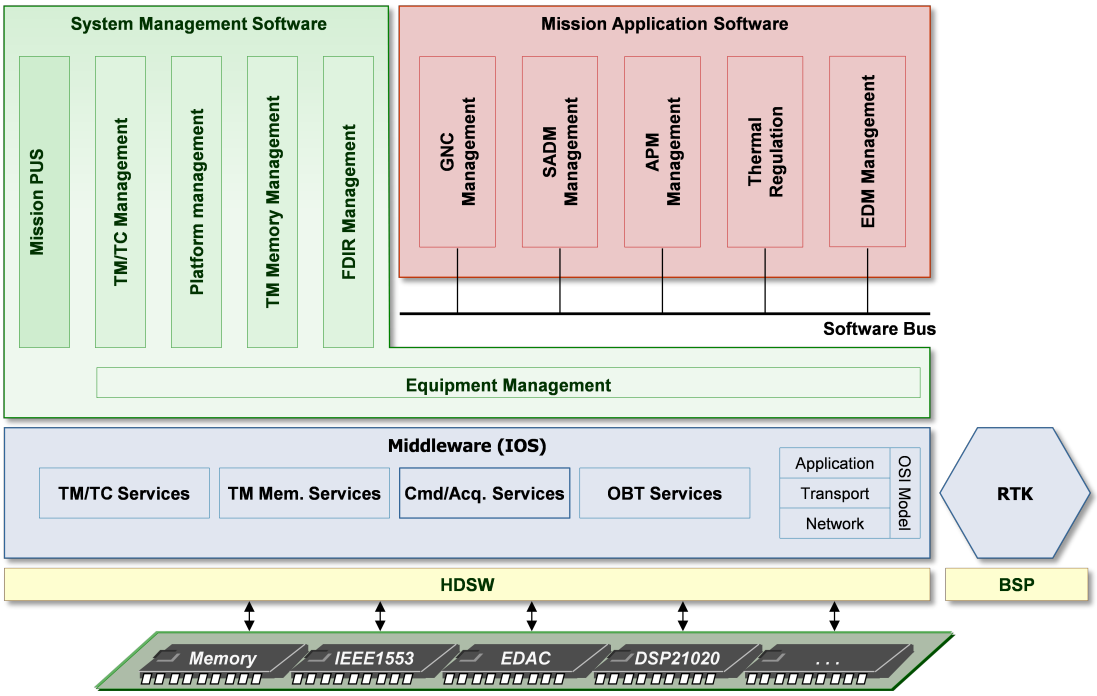

Figure 5.2: ExoMars TGO CSW Architecture.

Note: Figure 5.2 is from ExoMars TGO technical documentation.

As we can see on Figure 5.2, the TGO architecture is divided on four distinct layers. The yellow layer, on the bottom, has two different software components:

- **Hardware Dependent Software (HDSW) -** A component that provides the useful services for On-Board Computer (OBC) interfaces usage (e.g. Input/Output (I/O) links, M1553 buses, etc.). It is provided by the OBC manufacturer.
- **Board Support Package (BSP) -** A software linked with the Real-Time Kernel (RTK) that implements all the architecture specific features (e.g. PowerPC, SPARC, etc.) used by RTK.

The blue layer, is defined by the Middleware communication services and by the RTK.

- **Middleware, I/O Services** - a set of different communication services between the Network and the Application layer. Include services such as Telemetry (TM)/Telecommand (TC) exchange, an-board-time acquisition, etc.;
- **RTK -** provides real-time tasking features. The used was the OSTRALES [65] that is an Ada RTK for on-board satellite software applications.

The green layer, the System Management Software (SMS), includes a set of services common to several missions customised and adapted for which mission. The components included in this layer are the Package Utilization Standard (PUS) implementation, the Mission Timeline (MTL) and other scheduled variants, the Mil-Bus protocol, the spacecraft mode management and the on-board time maintenance. More detailed, the SMS is divided in the following components:

- **PUS-** Implementation of the PUS services according by ECSS-70-42B. For example, the initial checks of the telecommands received are performed on the SMS layer and are forwarded to the correct application to be processed;
- **TC/TM Management-** All the needed service for TM generation and TM processing;
- **TM Memory Management-** All the needed functions to manage the mass memory;
- **Platform Management -** Data handling services for all components such as memory, unit configuration, time and communication buses management;
- **Failure Detection, Isolation and Recovery (FDIR) -** The functionalities for failure detection and recovery;
- **Equipment Management -** Implements the functions to the platform operation;

The red layer includes the Mission Application Software (MAS) such as Guidance, Navigation and Control (GNC) software, Solar Array Drive Mechanism (SADM) control software, Antenna Pointing Mechanism (APM) control software, TR software and EDM Management application.

- **GNC Mode Management -** Implements the functions to manage all three modes (Guidance, Navigation, Control);
- **SADM Management -** Provides the Solar Array Drive Motor management services;
- **APM Management -** Provides the APM management services;
- **TR -** Provides the services for spacecraft thermal regulation, processing thermistances values and command heaters;
- **EDM Management -** Implements the functions to manage the payload equipment, the EDM;

5.5 TR Overview

In the operating spacecraft environment when a face is oriented to the sun the temperature on this face is very high and when is oriented to the shadow the temperature could be several negative degrees and therefore, we can have a huge difference of temperature between two different faces. Thus, to ensure that the spacecraft temperature are within bounds supported by the spacecraft hardware, the TR system function is control the temperature activating or deactivating the heaters.

ExoMars TGO has 105 thermal lines capable of heating the spacecraft equipment. A **thermal line** is a set of elements that regulate one unit. The thermal lines are organised as follow:

- **101** are composed of 3 thermistors and 2 heater lines (nominal and redundant). Controlled by TR application, that turn the heaters ON or OFF to maintain the temperature within thresholds;

- **4** are thermostat-regulated and always powered ON. This only can be commanded by ground-station;

Heater line is a heater commanded through the PCDU. ExoMars TGO has 105 nominal and 105 redundant heater lines. All heater lines are grouped into **Heater groups** that, when the a specific heater group is turned off all group heaters are also turned off. ExoMars TGO has 21 nominal and 21 redundant, each one having 5 heater lines of the same type, nominal and redundant respectively.
The thermal regulation algorithm applied by TR application to all 101 thermal lines consists on:

- Acquisition of the current resistance from the three thermistors and thermal line median temperature computation;
- Thermal control law application to determine the command to be issued based on the median temperature and the temperature thresholds;
- Issuing the heating command to the PCDU.

Farther, for FDIR purposes is verified to each thermal line if the temperature reaches one of the two limit levels that are detailed in FDIR design specification document. If any limit level is reached the FDIR component is responsible for the recovery actions.

5.6 Conclusion

This chapter presented an overview of ExoMars TGO mission, avionics system architecture and ExoMars TGO central software. For a spacecraft stable operation it is needed many different systems operating in parallel with distinct tasks. Five different systems divide the ExoMars CSW, each one with a distinct task.
TR software was the component validated and verified in this research work. The system is responsible for controlling all heating hardware keeping the spacecraft temperature always regulated.

Chapter 6

TR Detailed Design

6.1 Introduction

The goal of this research work was applying the SPARK formal verification to an Ada code already implemented and, understanding the code structure and how it was implemented turned the work more easy to be done. This chapter will present the detailed design of TR application, starting with the cyclic task that it performs and after the auxiliary components used to perform the cycle. The components will be presented in the cycle execution reverse order for in the main package, TR Manager, presentation his dependencies are lucid.
The information presented in this chapter is from TGO project documentation that are not public available.

6.2 TR Cyclic Task

TR application performs a cyclic task for all 101 thermal slots regulation at each $100ms$ ($10Hz$ frequency). Figure 6.1 presents the activity diagram for TR cyclic activities.

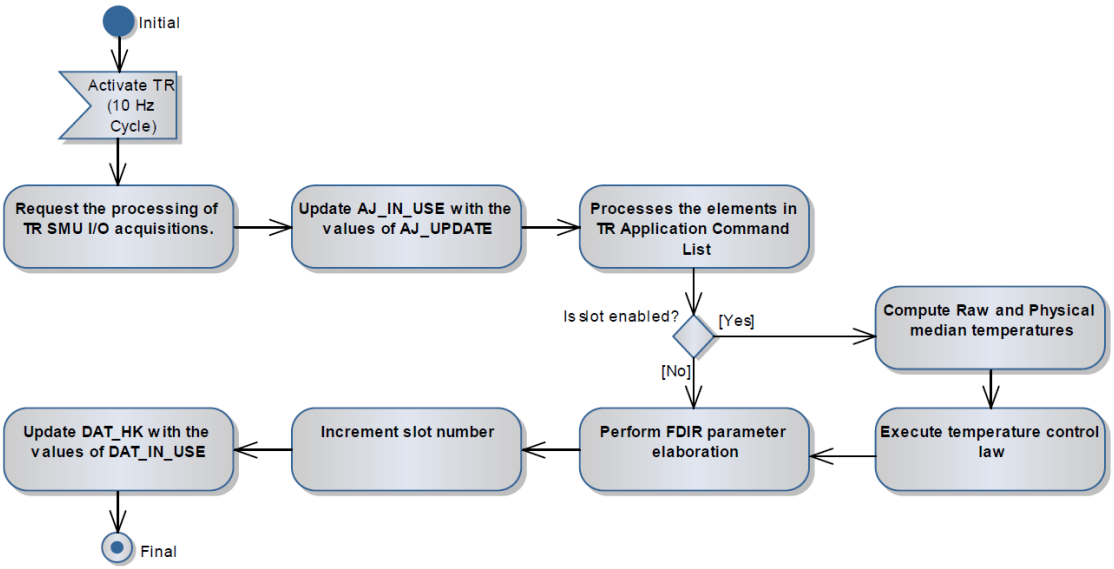

Figure 6.1: TR CYCLIC TASK.

The following activities are performed, by the same order, at each TR cycle:

- First, request the processing of TR SMU I/O acquisitions that are available on Data Pool and will be used by the TR application.
- Next, AJ_UPDATE records are copied to AJ_IN_USE;

- Before performing the Thermal Regulation application are processed the commands that must be executed during this cycle. This commands are stored in a list controlled by TR Command List application. See Section 6.6 for more detail;
- If the current slot is enabled then TR Regulation application, see Section 6.5, will perform the raw and physical median temperature from the 3 thermistors assigned to the slot and afterwards the thermal control law, turning on the heater if temperature drops the minimum threshold or off if drops the maximum threshold;
- Afterwards, FDIR application, see Section 6.4, will compute the median temperatures and check if any slot temperature has reached Level 1 or Level 2 temperatures;
- The slot number is incremented;
- Finally, DAT_IN_USE is copied to DAT_HK record to be available for housekeeping.

The activities above are performed in TR Manager package, that invokes functions of auxiliary packages to perform specific tasks. The elements in TR Application Command List are processed by TR Command List package, the thermal control law is applied by TR Regulation package and finally, the activities for FDIR purposes are part of TR Manager package, however, it will be considered as separate package. There are other two packages only for data purposes. The sections of this chapter are divided by the packages previously mentioned presenting the detailed design for each component.

6.3 TR Data Handling

The types, structures and variables that are used by components of TR application are declared and instantiated by two different packages, SDB and DATA.
First of all, it is important to understand the annotations attributed to the types. The meanings are as follows:

- **Adjustable (AJ)**;
- **DAT** stands for HouseKeeping (HK) information;

The **SDB** package contains AJ records that includes the adjustable parameters associated with the Satellite Database and TR component. It also has the initialisation values for those parameters.

The **DATA** package contains data records for the AJ and HK parameters. To ensure a consistent update of AJ parameters through each TR cycle, we have AJ_UPDATE that is a AJ record copy updated by ground station and AJ_IN_USE that is used during cycle execution. DAT_IN_USE is a HK copy updated through cycle that will be copied to DAT_HK when the cycle ends. It also contains all the definitions needed for the TR Command List implementation.

6.4 TR FDIR

Although TR FDIR is not a specific package, it is a component that should be presented apart of TR MANAGER. In Figure 6.2 we can see the procedures specifications.

Figure 6.2: TR FDIR.

- **PERFORM_FDIR_PARAMETER_ELABORATION -** This procedure performs FDIR parameter elaboration calling the next procedures;
- **COMPUTE_MEDIAN_TEMPERATURES -** This procedure determines the median temperatures in Celsius for a set of thermal line slots, which is determined by the slots interval passed by parameter. The computation for each slot is done by calling COMPUTE_RAW_MEDIAN_TEMPERATURE and COMPUTE_PHYSICAL_MEDIAN_TEMPERATURE of TR REGULATION (Section 6.5). An error report is sent if a math operation will cause an arithmetic error ($\log \leq 0$);
- **EVALUATE_LEVEL_1_STATUS -** This procedure evaluates the status for the FDIR level 1 check of median temperatures. It also evaluates a counter of the number of FDIR level 1 slot failures for each group;
- **EVALUATE_LEVEL_2_STATUS -** This procedure evaluates the status for the FDIR level 2 check of median temperatures. It also evaluates a counter of the number of FDIR level 2 slot failures.

6.5 TR REGULATION

This package express the essential procedures to perform the temperature regulation, called by TR MANAGER package. In Figure 6.3 we can see the procedures specifications.

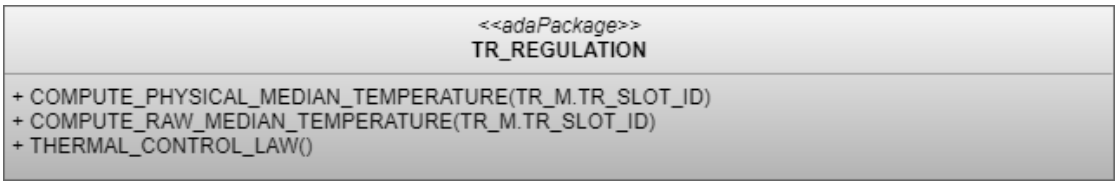

Figure 6.3: TR Regulation package.

- **COMPUTE_PHYSICAL_MEDIAN_TEMPERATURE -** This procedure converts the raw median temperature in Celsius according to the polynomial transfer function specific to each type of resistance (PT1000 or Beta G15K). If a math operation will cause an arithmetic error an invalid math operation ($\log \leq 0$) report shall be sent.
- **COMPUTE_RAW_MEDIAN_TEMPERATURE -** This procedure determines the raw median temperature from the 3 thermistors assigned to the SLOT_ID. The thermistors values are obtained from Platform Management application.
- **THERMAL_CONTROL_LAW -** This procedure implements the thermal control law, switching ON the heater if the temperature drops below the minimum threshold or OFF if crosses the maximum threshold.

6.6 TR COMMAND LIST

This package express the operations to manage the TR Command List, a list holding the commands to be performed. It is a linked list with pre-allocated nodes, that has the following attributes:

- **COUNTER -** Scheduler counter decremented at each iteration. When the value is 0, the element is retrieved from the list and the command is executed;
- **NEXT_POS -** Position of the next node in the list. If the list is not empty and the NEXT_POS is equals 0 means that this node is the last element in the list;
- **ELEMENT -** Element stored in the list;

In Figure 6.4 we can see the used global objects and procedures specifications.

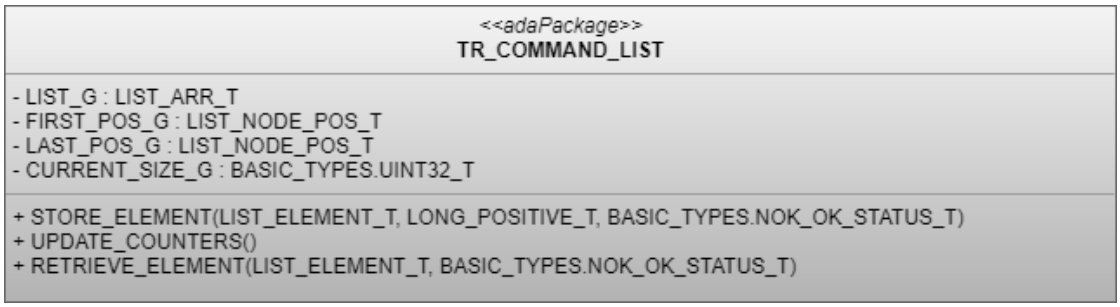

Figure 6.4: TR Command List package.

The package has extra attributes (Top of Figure 6.4) helpful to manage the linked list.

- **LIST_G -** Object that represents the command list;
- **FIRST_POS_G -** Object that represents the position of the first element in the command list;
- **LAST_POS_G -** Object that represents the position of the last element in the command list;
- **CURRENT_SIZE_G -** Object that represents the current size of the command list.

On bottom of Figure 6.4, are represented the three main operations performed to manage the command list. This three operations are the following:

- **STORE_ELEMENT -** This procedure adds an element to the list in the first free position next to the last element. If the list is not empty then STATUS, declared as output parameter, is set to OK else STATUS is set to NOK, informing if the element was correctly stored or not;
- **UPDATE_COUNTERS -** This procedure decreases, by one unit, the COUNTER attribute of each valid list element. It will iterates in NEXT_POS to NEXT_POS, between FIRST_POS_G and LAST_POS_G;
- **RETRIEVE_ELEMENT -** This procedure will retrieve the first element with COUNTER attribute equals 0. When a element is retrieved, the attributes are updated according with the following cases:
  - Element retrieved position is equal FIRST_POS_G;

- Element retrieved position is equal LAST_POS_G;
- Element retrieved position is between FIRST_POS_G and LAST_POS_G;
- Element retrieved position is equal FIRST_POS_G and LAST_POS_G;

As in STORE_ELEMENT, it has a STATUS parameter that is set to OK if a element is retrieved or to NOK if none element was find to retrieve. This procedure only retrieves one element at time. However, it is called by TR Manager since the STATUS value is different of NOK.

6.7 TR MANAGER

TR Manager purpose is to implement the cyclic task responsible for the execution of TR cycle, but also implements other functionalities as follows:

- Enabling and disabling TR function;
- Software and Functional Chain Validation (FCV) [1] context saving and restoration;
- FDIR parameter elaboration;

In Figure 6.5 we can see the procedures specifications.

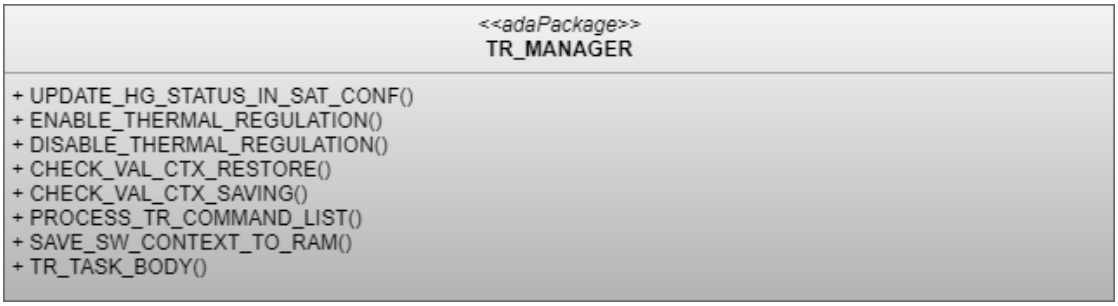

Figure 6.5: TR Manager package.

- **UPDATE_HG_STATUS_IN_SAT_CONF** - This procedure updates the heater groups status in spacecraft configuration table every time TR status change to enable or disable;
- **ENABLE_THERMAL_REGULATION** - When the TR function status changes from DISABLED to ENABLED it performs the following activities:
  - Set DAT variables values;
  - Set to ON PCDU heater groups according with the used flag;
  - Set all 210 PCDU heater lines to OFF;
- **DISABLE_THERMAL_REGULATION** - When the TR function status changes from ENABLED to DISABLED it sets to OFF all 210 PCDU heater lines and all 42 PCDU heater groups.
- **CHECK_VAL_CTX_RESTORE** - Checks if FCV is to be restored and after restores it if needed;
- **CHECK_VAL_CTX_SAVING** - Checks if FCV is to be saved and after saves it if needed;

[1] Ensuring that the implemented programs are verified against the customers requirements [66]

- **PROCESS_TR_COMMAND_LIST –** This procedure process the elements in TR Command List calling the procedures of TR Command List package. Calls UPDATE_COUNTERS to decrement the counter of the list elements and RETRIEVE_ELEMENT to remove the elements to be executed and finally executes them.
- **SAVE_SW_CONTEXT_TO_RAM –** This procedure saves the software context in RAM.
- **TR_TASK_BODY –** The main procedure of TR system. Its implementation is to perform the TR cyclic task according with what was presented in Section 6.2. This procedure are dependent of all procedures presented above in this package and all the auxiliary packages presented in Sections 6.4, 6.5 and 6.6.

6.8 Conclusion

This chapter presented the activities that are performed by TR cyclic task. After cyclic task analysis it was possible to divide the task in four different components and present for each one the detailed design considered for the implementation.

A good perception of how TR application was implemented and how is structured turns the next chapter, that presents how SPARK was applied in TR code, more easy to understand. It also was important to define how the analysis had to be performed, that is, in opposite direction of cycle execution proving all dependencies of TR Manager before the prove of this package.

Chapter 7

SPARK Analysis of TR Component

Verifying code already implemented and tested with a high reliable assurance was not an easy task. First, due to the fact that it was not implemented with SPARK analysis in mind, invalid features and false alarms detected by SPARK had to be bypassed following [27] that presents many approaches to prove Ada code already implemented. And other difficulty was that as code already had high levels of guarantee, all contracts annotated were orientated to find a minimal bug that has not been detected by the rigorous test performed instead of the main objectives of formal verification where the code has to be implemented to be compliant with the contracts specified.

The approach was to prove the TR system components in a opposite direction of cycle execution. This approach will avoid dependency problems when proof is targeted to TR Manager package in the main task procedure, assuming that all components are proved and his Post-Conditions are valid states. The main goal of this work was the analysis of information flow during programs execution and formal code verification to assure the safety and functional properties that are required for the TR application system.

7.1 Flow Analysis

The flow analysis main concerns is the data, it checks if the variables are correctly defined, instantiated or updated and, more important, it checks the correct information flow during the programs execution using the flow contracts available on SPARK toolset.

7.1.1 SPARK restrictions

As mentioned on 3.3.1, the first flow analysis phase is the detection of non-compatible SPARK features and in ExoMars TGO TR implementation, and others systems too, currently was used Ada Access Types to interface with data of other components. This use was hidden from SPARK analysis to the other analysis aspects be performed.

Following the approach in [27], the best practice to avoid that problem was the implementation of a new Ada package with SPARK analysis activated in the specification file and discarded on the package body. The procedures implemented in this package only executes the operations not valid that use the access types. In Listing 7.1 we can see the package specification and, one example of the many procedures that this package implements. The procedure EXECUTE_GET_TH_RESISTANCES interfaces with other package and the attribute **all** on SELF parameter is to access all attributes of this package.

```
1 package TR_M.CORE.ACCESS_TYPES with SPARK_Mode is
2
3   procedure EXECUTE_GET_TH_RESISTANCES(
```

```
    LINE_ID : in out TR_M.TR_TH_LINE_ID_T;
    RESIST_LINE : in out EQ_M.TH_M.EQ_TH_RESISTANCE_VAL_ARR_T);

end TR_M.CORE.ACCESS_TYPES;

package body TR_M.CORE.ACCESS_TYPES with SPARK_Mode => Off is

  procedure EXECUTE_GET_TH_RESISTANCES(
    LINE_ID : in out TR_M.TR_TH_LINE_ID_T;
    RESIST_LINE : in out EQ_M.TH_M.EQ_TH_RESISTANCE_VAL_ARR_T)
  is begin

    EQ_M.TH_M.EQ_TH_I.GET_TH_RESISTANCES(
        SELF => TR_M.CORE.DATA.TR_EQ_TH_I_PTR_G.all,
        THL_ID      => LINE_ID,
        RESISTANCES => RESIST_LINE);
  end EXECUTE_GET_TH_RESISTANCES;

end TR_M.CORE.ACCESS_TYPES;
```

Listing 7.1: Package for Access Types.

When a non-SPARK package declaration or body is included in a SPARK subprogram or package, the user has an obligation to ensure that the non-SPARK declaration is consistent. So the warnings for the use of this technique were ignored, assuming that the code are valid.

7.1.2 Flow Contracts

Flow contracts were used to prove the correct information flow during programs execution. The use of this contracts only was possible in programs that do not call the procedures implemented on package mentioned in 7.1.1 . In SPARK, the data used by non-SPARK code has to be listed on this contracts and, as this data is not valid SPARK code, the analysis will not be performed.
In Listing 7.2 we have an example of the Flow Contracts used in STORE_ELEMENT, procedure of TR Command List package.

```
procedure STORE_ELEMENT (ELEMENT_DATA : in LIST_ELEMENT_T;
             COUNTER      : in AO_MATH_TOOLBOX_MODULE.LONG_POSITIVE_T;
             STATUS       : out BASIC_TYPES.NOK_OK_STATUS_T) with SPARK_Mode,

  Global  => (
          Input     =>(MAX_LIST_SIZE_G),
          In_Out    =>(LIST_G, CURRENT_SIZE_G,
                       LAST_POS_G, FIRST_POS_G),
          Proof_In =>(IS_LIST_G_FULL_AUX_IN_STORE_ELEMENT,
                FREE_POS_IN_STORE_ELEMENT)
        ),

  Depends => (
          STATUS => (MAX_LIST_SIZE_G, CURRENT_SIZE_G,
                 LAST_POS_G,LIST_G ),
          LIST_G => (ELEMENT_DATA, COUNTER,
                 LAST_POS_G, CURRENT_SIZE_G, LIST_G ),
          CURRENT_SIZE_G => (CURRENT_SIZE_G, LAST_POS_G, LIST_G),
          LAST_POS_G     => (LAST_POS_G, CURRENT_SIZE_G, LIST_G),
          FIRST_POS_G    => (FIRST_POS_G, CURRENT_SIZE_G, LAST_POS_G, LIST_G)
        );
```

Listing 7.2: STORE_ELEMENT flow contracts.

The Global Contract present the most common annotations for global objects usage.

- **Input -** express that MAX_LIST_SIZE_G value is only read on this procedure;
- **In_Out -** express that, for example, CURRENT_SIZE_G initial value is read and finally updated;

- **Proof_In** - express the variables ghost declared only for proof;

In the example, it is important for Depends Contract consider the following cases:

- Express the dependencies for parameters specified as *out* or *in out*, in this case STATUS that depends of MAX_LIST_SIZE_G, CURRENT_SIZE_G and LAST_POS_G,LIST_G;
- Global objects annotated with *In_Out* or *Output* (LIST_G, CURRENT_SIZE_G, FIRST_POS_G and LAST_POS_G) in the Global Contract must have his dependencies specified;
- Sometimes when annotated as *In_Out* in Global Contract, the final value can depend of his initial value, for instance, CURRENT_SIZE_G initial value is incremented in this procedure;
- All global objects that the initial value is read, annotated with *Input* or *In_Out*, must be associated at least one variable in Depends Contract. MAX_LIST_SIZE_G, LIST_G, CURRENT_SIZE_G, LAST_POS_G and FIRST_POS_G are all associated in this Depends Contract.

7.2 Formal Code Verification

When planning the approach for formal code verification the first goal was subprograms safety. It is important to assure that no run-time exception will be launched. This kind of errors are detected by an automatic analysis performed by SPARK and sometimes the proof of their absence is not automatic and only possible using the contracts specifying the necessary properties to help the SPARK analysis.

The proof is the more complex activity when using SPARK analysis. Sometimes contracts are not proved automatically, and SPARK annotations are necessary in order to prove programs bodies consistency and help SPARK proving most complex properties. The use of assertions to prove program consistency, Loop Invariants and Variant to prove loop consistency and Ghost Code to produce properties not available for SPARK was useful for almost all programs proved.

This section is divided by four subsections, each one corresponding to one component of those identified in Chapter 6. In each subsection is presented the proof objectives for the component and examples of the work performed with SPARK. In order to avoid the document extension only one example will be presented for each component and this example only presents the useful code (Full code is available in [67]).

7.2.1 TR FDIR

TR FDIR main procedure is PERFORM_FDIR_PARAMETER_ELABORATION that calls auxiliary COMPUTE_MEDIAN_TEMPERATURES, EVALUATE_LEVEL_1_STATUS and EVALUATE_LEVEL_2_STATUS, for more info see Section 6.4. It will be presented the work in COMPUTE_MEDIAN_TEMPERATURES. This procedure determines the median temperatures for a set of thermal lines passed by parameter. In Listing 7.3, we can see the procedure specification and the contracts included.

```
1 procedure COMPUTE_MEDIAN_TEMPERATURES (INITIAL_SLOT_ID, FINAL_SLOT_ID : in TR_M.TR_SLOT_ID_T)
2   with SPARK_Mode,
3   Pre  => INITIAL_SLOT_ID <= FINAL_SLOT_ID,
4   Post => SLOT_ID_IN_COMPUTE_MEDIAN_TEMPERATURES = FINAL_SLOT_ID
5       and
```

```
6       (if SEND_ATM_LOG_ZERO_OR_NEG_EVENT_GHOST then EXECUTED_P_A_T_I_M_O)
7   ;
```

Listing 7.3: COMPUTE_MEDIAN_TEMPERATURES procedure specification.

The contracts are explained as follows:

- Line 3: This is a Pre-Condition meaning that to guarantee a valid interval the final value has to be bigger than initial;
- Line 4: This is a Post-Condition. First to guarantee that the procedure will iterate all the interval, SLOT_ID_IN_COMPUTE_MEDIAN_TEMPERATURES ghost global variable, that will be incremented at each iteration, in the end has to be equal to FINAL_SLOT_ID. And, in the second case, if calling the methods of TR Regulation package to compute the median temperatures caused an arithmetic error then EXECUTED_P_A_T_I_M_O must be TRUE, to guarantee that EXECUTE_PUSH_ATM_TR_INVALID_MATH_OPERATION, defined in Access Types package, was executed.

Next, in Listing 7.4 we can see the procedure body and all the annotations included (See [67] for full code).

```
1   SLOT_ID := INITIAL_SLOT_ID;
2
3   if (SLOT_ID <= FINAL_SLOT_ID) then DO_LOOP := TRUE; end if;
4
5   while DO_LOOP = TRUE loop
6
7     pragma Loop_Variant(Decreases => FINAL_SLOT_ID - SLOT_ID);
8     pragma Loop_Invariant(SLOT_ID in INITIAL_SLOT_ID .. FINAL_SLOT_ID);
9     pragma Loop_Invariant(if WAS_AUX_SEND_ATM_EVENT_TRUE then SEND_ATM_LOG_ZERO_OR_NEG_EVENT = TRUE);
10
11    TR_M.CORE.TR_REGULATION.COMPUTE_RAW_MEDIAN_TEMPERATURE (SLOT_ID => SLOT_ID);
12    TR_M.CORE.TR_REGULATION.COMPUTE_PHYSICAL_MEDIAN_TEMPERATURE ( SLOT_ID => SLOT_ID,
13        SEND_ATM_LOG_ZERO_OR_NEG_EVENT => AUX_SEND_ATM_EVENT);
14
15    SEND_ATM_LOG_ZERO_OR_NEG_EVENT := SEND_ATM_LOG_ZERO_OR_NEG_EVENT or AUX_SEND_ATM_EVENT;
16
17    SET_GHOST_SEND_ATM_EVENT_TRUE;
18
19    if (SLOT_ID = FINAL_SLOT_ID) then DO_LOOP := FALSE; else SLOT_ID := SLOT_ID + 1; end if;
20  end loop;
21
22  SLOT_ID_IN_COMPUTE_MEDIAN_TEMPERATURES := SLOT_ID;
23  pragma Assert(SLOT_ID_IN_COMPUTE_MEDIAN_TEMPERATURES = FINAL_SLOT_ID);
24
25  pragma Assert(if WAS_AUX_SEND_ATM_EVENT_TRUE then SEND_ATM_LOG_ZERO_OR_NEG_EVENT = TRUE);
26  SEND_ATM_LOG_ZERO_OR_NEG_EVENT_GHOST := SEND_ATM_LOG_ZERO_OR_NEG_EVENT;
27
28  if (SEND_ATM_LOG_ZERO_OR_NEG_EVENT = TRUE) then
29    TR_M.CORE.ACCESS_TYPES.EXECUTE_PUSH_ATM_TR_INVALID_MATH_OPERATION(LOG_ZERO_OR_NEG_EVENT);
30
31    pragma Assert(EXECUTED_P_A_T_I_M_O);
32  end if;
```

Listing 7.4: COMPUTE_MEDIAN_TEMPERATURES procedure body.

The procedure implements a while loop that iterates all the slots defined by the interval and performs the median temperature computation for each specific slot. At the end, checks if an arithmetic error results for this computation and executes EXECUTE_PUSH_ATM_TR_INVALID_MATH_OPERATION procedure of Access Types (Line 30).
The annotations included are explained as follows:

- Line 7 : Loop Variant means that the difference between FINAL_SLOT_ID and SLOT_ID must decrease at each iteration to guarantee the loop termination;

- Line 8 : Loop Invariant to guarantee that SLOT_ID is always a value within the parameter thresholds;
- Line 9 : Loop Invariant to preserve the disjunction rule, in this case $0 \wedge 1 = 1$ at each iteration. When WAS_AUX_SEND_ATM_EVENT_TRUE is True means that at least in one iteration was launched an arithmetic error and SEND_ATM_LOG_ZERO_OR_NEG_EVENT is also True.
- Line 17 : Ghost Procedure that updates WAS_AUX_SEND_ATM_EVENT_TRUE, Ghost variable, to True when AUX_SEND_ATM_EVENT is True;
- Line 22 : Ghost variable SLOT_ID_IN_COMPUTE_MEDIAN_TEMPERATURES is updated to final value of SLOT_ID. Variable used to verify the Post-Condition;
- Line 23 : Assertion to check if SLOT_ID_IN_COMPUTE_MEDIAN_TEMPERATURES is equal to the last slot of the interval;
- Line 25 : Assertion to enforce the condition defined on Loop Invariant of Line 9 after the cycle execution;
- Line 26 : Ghost Variable SEND_ATM_LOG_ZERO_OR_NEG_EVENT_GHOST is updated to the value of SEND_ATM_LOG_ZERO_OR_NEG_EVENT. Variable used to verify the Post-Condition;
- Line 29 : Execution of PUSH_ATM_TR_INVALID_MATH_OPERATION that now is defined on Access Types package;
- Line 31 : Assertion to check if Ghost Variable EXECUTED_P_A_T_I_M_O is equals True. This means that EXECUTE_PUSH_ATM_TR_INVALID_MATH_OPERATION was executed.

7.2.2 TR REGULATION

TR Regulation package performs three distinct procedures to be used on TR Manager for thermal regulation activities (See Section 6.5 for more info). These three procedures were not complex enough to use all SPARK features but, COMPUTE_RAW_MEDIAN_TEMPERATURE is a good example where the contracts annotated shows that this approach can totally be an alternative to software testing.

This procedure computes the raw median temperature from the three thermistors assigned to thermal line, identified the parameter SLOT_ID. Considering R_{median} the thermistor with median temperature and R_1 e R_2 the other two thermistors of thermal line, the approach of this function is identify R_{median} as median temperature if $R_1 \leq R_{median} \leq R_2 \vee R_2 \leq R_{median} \leq R_1$.

The unit tests performed for this procedure checked the following different scenarios:

- **Scenario 1:** $R_1 < R_{median} < R_2$
- **Scenario 1:** $R_2 < R_{median} < R_1$
- **Scenario 1:** $R_1 = R_{median} = R_2$
- **Scenario 1:** $R_1 < (R_{median} = R_2)$
- **Scenario 1:** $R_2 < (R_{median} = R_1)$
- **Scenario 1:** $(R_1 = R_{median}) < R_2$

- **Scenario 1:** $(R_2 = R_{median}) < R_1$

In COMPUTE_RAW_MEDIAN_TEMPERATURE specification, see Listing 7.5, we can see how one simple condition annotated as Post-Condition can replace all scenarios presented above and it is not necessary the use of extra tools to prove that code is correct.

```
1 procedure COMPUTE_RAW_MEDIAN_TEMPERATURE (SLOT_ID : in TR_M.TR_SLOT_ID_T)
2   with SPARK_Mode,
3   Post => R_MEDIAN in RO1 .. RO2 or R_MEDIAN in RO2 .. RO1;
```

Listing 7.5: COMPUTE_RAW_MEDIAN_TEMPERATURE procedure specification.

The Post-Condition, Line 3, simulates the condition mentioned above, $R_1 \leq R_{median} \leq R_2 \vee R_2 \leq R_{median} \leq R_1$. This condition assimilates all the possible scenarios that was checked on software unit testing. R_MEDIAN, RO1 and RO2 are Ghost variables updated when the thermistor with the median temperature is identified.

Next, in Listing 7.6 we can see the procedure body and all annotations included. It is a conditional program that compares the temperatures of the three different thermistors and the verification process is the same for all thus, in this example, only will be presented the case when the median is R2 (See [67] for full code).

```
1  LINE_ID := TR_M.CORE.UTILS.TR_SLOT_TO_LINE_C (SLOT_ID);
2
3  pragma Assert(LINE_ID in 1 .. 105);
4  pragma Assert(  if SLOT_ID in 087 .. 095 then LINE_ID = SLOT_ID + 1
5              elsif SLOT_ID in 096 .. 101 then LINE_ID = SLOT_ID + 2
6            else LINE_ID = SLOT_ID);
7
8  TR_M.CORE.ACCESS_TYPES.EXECUTE_GET_TH_RESISTANCES(LINE_ID, RESIST_LINE);
9
10 if ( ( (RESIST_LINE (EQ_M.TH_M.R1) <= RESIST_LINE (EQ_M.TH_M.R2)) and then
11   (RESIST_LINE (EQ_M.TH_M.R2) <= RESIST_LINE (EQ_M.TH_M.R3)) ) or else
12   ( (RESIST_LINE (EQ_M.TH_M.R3) <= RESIST_LINE (EQ_M.TH_M.R2)) and then
13   (RESIST_LINE (EQ_M.TH_M.R2) <= RESIST_LINE (EQ_M.TH_M.R1)) ) ) then
14
15     TR_M.CORE.DATA.DAT_IN_USE_G.TH.CSW_SLOT_RESIST_MEDIAN (SLOT_ID) := RESIST_LINE (EQ_M.TH_M.R2);
16
17     R_MEDIAN := RESIST_LINE(EQ_M.TH_M.R2);
18     RO1 := RESIST_LINE(EQ_M.TH_M.R1);
19     RO2 := RESIST_LINE(EQ_M.TH_M.R3);
20
21     pragma Assert(RESIST_LINE(EQ_M.TH_M.R2) in RESIST_LINE(EQ_M.TH_M.R1) .. RESIST_LINE(EQ_M.TH_M.R3)
22              or
23              RESIST_LINE(EQ_M.TH_M.R2) in RESIST_LINE(EQ_M.TH_M.R3) .. RESIST_LINE(EQ_M.TH_M.R1));
24
25     -- (...)
26
27 end if;
```

Listing 7.6: COMPUTE_RAW_MEDIAN_TEMPERATURE procedure body.

The annotations are explained as follows:

- Line 3 : Assertion that checks if LINE_ID is in the correct defined range;
- Line 4 : Assertion that checks if the LINE_ID value is the correct for the SLOT_ID, according with content tables available on TR requirements documentation;
- Line 8 : Execution of GET_TH_RESISTANCES that now is defined on Access Types package;

- Line 17 : Update the global Ghost Variable R_MEDIAN to the temperature in the thermistor R2. Used on Post-Condition;
- Lines 18 and 19 : Update the global Ghost Variables RO1 and RO2 to the remaining thermistors temperature;

7.2.3 TR COMMAND LIST

In Section 6.6 was presented a set of attributes for the Command List manipulation. To prove, in addition to functional properties proving, the full correctness of Command List manipulation there are a set of states that has to be valid before and after a list updating procedure execution. Following the work performed in [68], to achieve the full correctness was implemented a Ghost function, see Listing 7.7, that returns a Boolean value that indicates if the list is in a correct state or not. This function is a predicate or a lemma with a set of conditions that state the compatibility between the Command List and a logical view.

```
1  function INVARIANT return Boolean is
2  (
3    if CURRENT_SIZE_G = 0 then
4      IS_CURRENT_SIZE_IN_BOUNDS and
5      FIRST_POS_G = 0 and LAST_POS_G = 0 and
6      (for all I in VALID_POSITIONS'Range => VALID_POSITIONS(I) = 0) and
7      (for all I in LIST_G'Range => LIST_G(I).COUNTER = 0 and LIST_G(I).NEXT_POS = 0)
8
9    elsif CURRENT_SIZE_G = 1 then
10     IS_CURRENT_SIZE_IN_BOUNDS  and
11     FIRST_POS_G in LIST_G'Range and LAST_POS_G in LIST_G'Range and FIRST_POS_G = LAST_POS_G and
12     LIST_G(FIRST_POS_G).NEXT_POS = 0 and
13     (for all I in LIST_G'Range => (if I /= FIRST_POS_G and I /= LAST_POS_G
14         then LIST_G(I).COUNTER = 0 and LIST_G(I).NEXT_POS = 0)) and
15     VALID_POSITIONS(VALID_POSITIONS'First) = FIRST_POS_G and
16     (for all I in CURRENT_SIZE_G + 1 .. VALID_POSITIONS'Last => VALID_POSITIONS(I) = 0)
17
18   elsif CURRENT_SIZE_G > 1 then
19     IS_CURRENT_SIZE_IN_BOUNDS and
20     (if CURRENT_SIZE_G < MAX_LIST_SIZE_G then (for some I in LIST_G'Range =>
21     LIST_G(I).COUNTER = 0 and LIST_G(I).NEXT_POS = 0 and I /= LAST_POS_G and I /= FIRST_POS_G)) and
22     FIRST_POS_G in LIST_G'Range and LAST_POS_G in LIST_G'Range and FIRST_POS_G /= LAST_POS_G and
23     VALID_POSITIONS_OK(FIRST_POS_G, 1) and VALID_NOT_REACHABLE and
24     VALID_POSITIONS(VALID_POSITIONS'First) = FIRST_POS_G and VALID_POSITIONS(CURRENT_SIZE_G) =
           LAST_POS_G and
25     LIST_G(LAST_POS_G).NEXT_POS = 0 and LIST_G(LAST_POS_G).COUNTER >= 0
26 ) with Ghost;
```

Listing 7.7: Command List Invariant.

First, it is important to know that VALID_POSITIONS is a Ghost global array with the same range as LIST_G containing the valid positions (positions with elements) of the list ordered, starting with the first position until the position equals to CURRENT_SIZE_G. There are three different conditions for list state specification explained as follows:

- Command List empty (CURRENT_SIZE_G = 0):
  - Line 4 : IS_CURRENT_SIZE_IN_BOUNDS is a Ghost function that indicates if CURRENT_SIZE_G does not crossed the MAX_LIST_SIZE_G limit;
  - Line 5 : FIRST_POS_G and LAST_POS_G equals 0, value out of list range, meaning that does not exist first position neither last;
  - Line 6 : All elements in VALID_POSITIONS are equals 0;

  - Line 7 : All COUNTER and NEXT_POS attributes in LIST_G are equals 0;

- Command List with one element (CURRENT_SIZE_G = 1):
  - Line 10 : IS_CURRENT_SIZE_IN_BOUNDS is a Ghost function that indicates if CURRENT_SIZE_G does not crossed the MAX_LIST_SIZE_G limit;
  - Line 11 : FIRST_POS_G and LAST_POS_G must be the same value because only exists one element and the value must be in LIST_G range meaning that is a valid position;
  - Line 12 : NEXT_POS of the element must be 0 because does not exist more elements;
  - Line 13 : For all elements, except the element on the first position (or last), the NEXT_POS and COUNTER attributes must be 0;
  - Line 15 : The first position of VALID_POSTIONS must be equals to FIRST_POS_G;
  - Line 16 : The remaining elements of VALID_POSTIONS must be equals 0;
- Command List with more than one element (CURRENT_SIZE_G > 1):
  - Line 19 : IS_CURRENT_SIZE_IN_BOUNDS is a Ghost function that indicates if CURRENT_SIZE_G does not crossed the MAX_LIST_SIZE_G limit;
  - Line 20 : If CURRENT_SIZE_G is strictly lower than MAX_LIST_SIZE_G, must exist at least one element with COUNTER and NEXT_POS attributes equals 0. And this element cannot be equal to FIRST_POS_G and LAST_POS_G;
  - Line 23 : VALID_POSITIONS_OK is a Ghost recursive function that checks if elements in VALID_POSITIONS are according with the valid elements in LIST_G. In the same line, VALID_NOT_REACHABLE checks if not valid positions in LIST_G has the attributes equals 0;
  - Line 24 : First position of VALID_POSTIONS must be equal to FIRST_POS_G and the position equals to CURRENT_SIZE_G must be equal to LAST_POS_G;
  - Line 25 : NEXT_POS of LAST_POS_G must be equals zero and COUNTER can be 0 or bigger.

The correctness criteria of all the operations on Command List (LIST_G) is stated with respect to this logical view. The operation are correct if their actions do not alter the compatibility between the logical view and the Command List in concrete. In other words, the Ghost counterpart of the Command List is a well behaved implementation that does not suffer from the concrete contingency (and apparent limitations) that SPARK programs must follow nor the expected to be efficient. The Ghost code is defined in way that mimics the logical behaviour of Command List. The correctness criteria can be paraphrased as both logical and SPARK implementation on the list work the same way.

Three updates are made under Command List, but here it is presented the verification of UPDATE_COUNTERS procedure, considering the logical view and its required functional properties. In Listing 7.8 we can see the procedure specification.

```
1 procedure UPDATE_COUNTERS
2   with SPARK_Mode,
3   Pre => INVARIANT,
4   Post =>(for all I in VALID_POSITIONS'First..CURRENT_SIZE_G=> (if LIST_G'Old(I).COUNTER /= 0 then
```

```
5                                        LIST_G(I).COUNTER = LIST_G'Old(I).COUNTER - 1))
6       and
7       INVARIANT
8 ;
```

Listing 7.8: UPDATE_COUNTERS procedure specification.

The contracts are explained as follows:

- Line 3 : Pre-Condition to ensure that INVARIANT is valid at the beginning;
- Line 4 : Post-Condition that checks for each position in VALID_POSITIONS if in the old list at this position the COUNTER attribute was different than 0 then in the output list the COUNTER attribute must be lower than oldest value;
- Line 7 : Other condition of Post-Condition to ensure that INVARIANT is valid in the end;

UPDATE_COUNTERS implements a loop to iterate only the positions with elements and decrements his COUNTER attribute merely if the list is not empty (CURRENT_SIZE_G not 0). In Figure 7.1 there is an example of a possible loop sequence, considering LAST_POS_G = 4 and CURRENT_SIZE_G = 4.

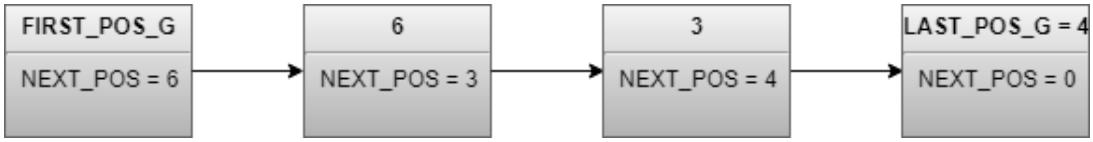

Figure 7.1: UPDATE_COUNTERS loop iterations workflow.

Next, in Listing 7.9, is presented the UPDATE_COUNTERS procedure body with all annotations. As mentioned before the results here were not the best and, in this example some conditions were assumed valid to prove other conditions (See [67] for full code).

```
1  if (CURRENT_SIZE_G /= 0) then
2
3    pragma Assert(FIRST_POS_G in LIST_G'Range);
4    pragma Assert(INVARIANT);
5
6    POS_ITERATOR := FIRST_POS_G;
7
8    loop
9      if (LIST_G (POS_ITERATOR).COUNTER /= 0) then
10     LIST_G (POS_ITERATOR).COUNTER := LIST_G (POS_ITERATOR).COUNTER - 1;
11     end if;
12
13     pragma Loop_Invariant(POS_ITERATOR in LIST_G'Range);
14     pragma Loop_Invariant(for all I in VALID_POSITIONS'First .. LOOP_COUNTER =>
15       (if LIST_G'Loop_Entry(I).COUNTER /= 0 then LIST_G(I).COUNTER = LIST_G'Loop_Entry(I).COUNTER -
             1));
16     pragma Loop_Invariant(INVARIANT);
17
18     exit when (POS_ITERATOR = LAST_POS_G);
19
20     POS_ITERATOR := LIST_G (POS_ITERATOR).NEXT_POS;
21     pragma Assume(POS_ITERATOR in LIST_G'Range);
22
23     pragma Assume(LOOP_COUNTER < LIST_NODE_POS_T'Last);
24     LOOP_COUNTER := LOOP_COUNTER + 1;
25   end loop;
26
27 end if;
28
```

```
29 pragma Assume(LOOP_COUNTER = CURRENT_SIZE_G);
```

Listing 7.9: UPDATE_COUNTERS procedure body.

The annotations are explained as follows:

- Line 3 : Assertion to assure that FIRST_POS_G is in LIST_G range avoiding a range violation in Lines 9 and 10 at first iteration;
- Line 4 : There are no alterations one the list before, however, as it is a conditional program, with only one condition, this assertion is to keep the info about INVARIANT consistent to SPARK;
- Line 13 : Loop Invariant to keep the condition of Line 3 consistent for all the remaining iterations;
- Line 14 : This Loop Invariant assures the condition defined in the Post-Condition, however, it only checks the elements until POS_ITERATOR;
- Line 16 : Loop Invariant assuring INVARIANT at all iterations. Not proved but, with this annotation the Post-Condition is proved;
- Line 21 : The range of NEXT_POS is 0 to MAX_LIST_SIZE_G and, when equals 0 it is out of LIST_G range. In order to maintain the properties of Line 13, SPARK cannot prove that will never be 0. This is a property that must be assured before at Line 20 or in a Pre-Condition and it was not reached here. This Assume covered this problem, for now.
- Line 23 : To SPARK assume that LOOP_COUNTER is strictly lower than maximum value to prove that the incrementation on Line 24 does not result on overflow;
- Line 29 : Assuming LOOP_COUNTER equals to CURRENT_SIZE_G to the prove of Post-Condition considering the Loop Invariant on Line 14;

It is important to highlight that the correctness properties are rather technical to prove UPDATE_COUNTERS and the ramainig procedures that updates the data-structure. As mentioned before, the correctness criteria is state as a compatibility predicate between the Command List and the Ghost logical view defined.

7.2.4 TR MANAGER

In TR Manager is the main package where the main cyclic task, TR_TASK_BODY, is defined to control the TR application. For TR_TASK_BODY prove, the last in this case, all Post-Conditions of subprograms that this procedure is dependent were assumed proved, even the INVARIANT of TR Command List, and the only concern was guarantee a state that satisfies the condition required by this subprograms.

In this package are defined other auxiliary subprograms to TR_TASK_BODY and the case presented here is part of one of the most relevant subprograms. ENABLE_THERMAL_REGULATION updates all the needed values when the TR function status change to ENABLED. Almost all updates performed in this subprogram are trivial and straight thus, it will be discarded in this presentation. The update presented here is set the PCDU flag of each unit according the used flag.

To perform this update, there is an array declared with 42 positions that will save the update value for each Unit of every group ($21Groups * 2Units$). In Figure 7.2, we can see the representation of how the array position corresponds to a unit of a specific group.

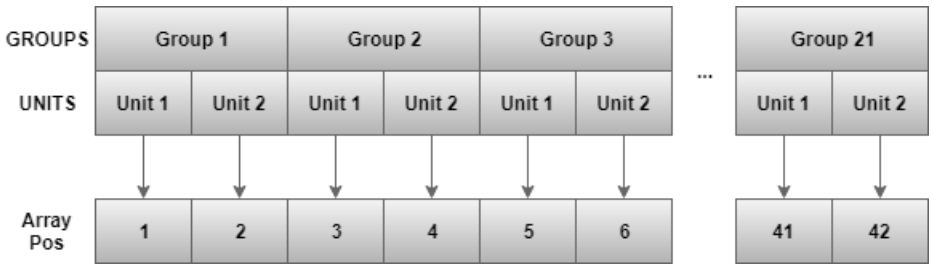

Figure 7.2: Array Positions.

In Listing 7.10 is represented the procedure specification.

```
procedure ENABLE_THERMAL_REGULATION with SPARK_Mode,
  Pre  => ENABLE_THERMAL_REGULATION_ATTRIBUTIONS_OK,
  Post => ENABLE_THERMAL_REGULATION_ATTRIBUTIONS_OK
;
```

Listing 7.10: ENABLE_THERMAL_REGULATION procedure specification.

First, two Ghost arrays with 42 positions, one (USED_OR_NOT_USED_ARRAY) saving the currently used flag (USED, NOT_USED) for each unit and other (OPERATIONAL_OR_OFF_ARRAY) saving the attribution (OPERATIONAL, OFF), were created. The same positions in both arrays coincide with the same unit. The Pre and Post-Conditions is a condition defined in a Ghost function, ENABLE_THERMAL_REGULATION_ATTRIBUTIONS_OK, that checks for all elements in the Ghost arrays if the flag is USED then OPERATIONAL else if NOT_USED then OFF.
Note: Although the Post-Condition is more complex, for this example consider only this condition (See [67] for full code).

Next, in Listing 7.11 the code that performs the attribution and all annotations (See [67] for full code). There is a **For** loop that will iterate for all 21 Groups and inside another **For** loop iterating the two Units with a conditional statement checking the currently used flag.

```
for GROUP_ID in TR_M.TR_HG_ID_T'RANGE loop
  for UNIT_ID in TR_M.TR_UNIT_ID_T'RANGE loop

    if (TR_M.CORE.UTILS.IS_UNIT_CURRENTLY_USED (GROUP_TYPE => GROUP_UNIT_TYPE, UNIT_ID => UNIT_ID) =
        PM_M.SAT_CONF_M.USED) then
    -- Command PCDU heater group to switch ON
    -- Set UNIT status to OPERATIONAL
    SET_ARRAY_POS_USED(HG_SATCONF_UPDATE_INDEX);
    OPERATIONAL_OR_OFF_ARRAY(HG_SATCONF_UPDATE_INDEX) := HG_SATCONF_UPDATE_ARR_42_G (
        HG_SATCONF_UPDATE_INDEX).UNIT_STATUS;

    else
    -- Command PCDU heater group to switch OFF
    -- Set UNIT status to OFF
    SET_ARRAY_POS_NOT_USED(HG_SATCONF_UPDATE_INDEX);
    OPERATIONAL_OR_OFF_ARRAY(HG_SATCONF_UPDATE_INDEX) := HG_SATCONF_UPDATE_ARR_42_G (
        HG_SATCONF_UPDATE_INDEX).UNIT_STATUS;

    end if;
    pragma Assume(if UNIT_ID = 1 then HG_SATCONF_UPDATE_INDEX = HG_SATCONF_UPDATE_ARR_42_INDEX_T(2 *
        (GROUP_ID - 1) + 1)
           elsif UNIT_ID = 2 then HG_SATCONF_UPDATE_INDEX = HG_SATCONF_UPDATE_ARR_42_INDEX_T(2 *
               GROUP_ID + 1));

```

```
20      if (HG_SATCONF_UPDATE_INDEX < HG_SATCONF_UPDATE_ARR_42_INDEX_T'LAST) then
21        HG_SATCONF_UPDATE_INDEX :=  HG_SATCONF_UPDATE_INDEX + 1;
22      end if;
23
24      pragma Loop_Invariant(if USED_OR_NOT_USED_ARRAY(HG_SATCONF_UPDATE_INDEX'Loop_Entry) = PM_M.
            SAT_CONF_M.USED then OPERATIONAL_OR_OFF_ARRAY(HG_SATCONF_UPDATE_INDEX'Loop_Entry) = PM_M.
            SAT_CONF_M.OPERATIONAL
25                  else OPERATIONAL_OR_OFF_ARRAY(HG_SATCONF_UPDATE_INDEX'Loop_Entry) = PM_M.SAT_CONF_M.
                        OFF);
26    end loop;
27    pragma Assume(HG_SATCONF_UPDATE_INDEX = HG_SATCONF_UPDATE_ARR_42_INDEX_T(2 * GROUP_ID + 1));
28    pragma Loop_Invariant(for all I in HG_SATCONF_UPDATE_INDEX - 2 .. HG_SATCONF_UPDATE_INDEX - 1 =>
29                 (if USED_OR_NOT_USED_ARRAY(I) = PM_M.SAT_CONF_M.USED then OPERATIONAL_OR_OFF_ARRAY(I) =
                        PM_M.SAT_CONF_M.OPERATIONAL
30                  else OPERATIONAL_OR_OFF_ARRAY(I) = PM_M.SAT_CONF_M.OFF));
31  end loop;
```

Listing 7.11: ENABLE_THERMAL_REGULATION procedure body.

The annotations are explained as follows:

- Line 7 : Ghost program to set USED_OR_NOT_USED_ARRAY in HG_SATCONF_UPDATE_INDEX position to USED;
- Line 8 : Set OPERATIONAL_OR_OFF_ARRAY in position HG_SATCONF_UPDATE_INDEX to the UNIT_STATUS attributed;
- Line 13: Ghost program to set USED_OR_NOT_USED_ARRAY in HG_SATCONF_UPDATE_INDEX position to NOT_USED;
- Line 14 : Set OPERATIONAL_OR_OFF_ARRAY in HG_SATCONF_UPDATE_INDEX position to the UNIT_STATUS attributed;
- Line 24 : Loop Invariant that proves the correct attribution to this specific Unit. The condition is the same as ENABLE_THERMAL_REGULATION_ATTRIBUTIONS_OK and helps SPARK proving the Post-Condition.
- Line 27 : Loop Invariant that proves the correct attribution of the two Units to this Group. As Line 24, the condition is the same as ENABLE_THERMAL_REGULATION_ATTRIBUTIONS_OK and helps SPARK proving the Post-Condition;
- Lines 17, 27 : SPARK could not prove the Loop Invariants, due to the low knowledge about the array positions associations as presented in Figure 7.2. It is known that the positions in the array are given by the Equation 7.1 and if SPARK assumes this conditions, that are completely valid and does not have influence in the code behaviour, the Loop Invariants of Lines 24 and 27 will be proved and therefore the Post-Condition too.

$$\begin{cases} 2 * (Group - 1) + 1, & \text{if } Unit = 1 \\ 2 * Group + 1, & \text{if } Unit = 2 \end{cases} \quad (7.1)$$

7.3 Final Results

The final results of TR system SPARK analysis are presented in Table 7.1. The table shows the packages results in this work only with CVC4 analysis, that is the GNATProve default SMT prover. Other provers as Z3 and Alt-Ergo were used for helping the proof of not proved properties with CVC4, nevertheless the results were always the same.

SPARK Analysis results	Total	Flow	Interval	Provers	Unproved
Data Dependencies	-	-	-	-	-
Flow Dependencies	-	-	-	-	-
Initialization	247	245	-	-	2
Non-Aliasing	-	-	-	-	-
Run-time Checks	132	-	5	114 (CVC4)	13
Assertions	111	-	-	100 (CVC4)	11
Functional Contracts	39	-	-	37 (CVC4)	2
LSP Verification	-	-	-	-	-
Total	529	245 (46%)	5 (1%)	251 (47%)	28 (5%)

Table 7.1: GNATProve Report.

Regarding Flow Analysis, the automatic check results were considerably good, in 247 variables initialization checks only 2 were not proved, however, were considerer as SPARK false alarm since it will not influence the remaining code and no case of alias occurrence was detected, showing that alias cases is really an issue to avoid that was considerer in ExoMars CSW development. In the release used in this work, SPARK 2014 Discovery, GNATProve does not report yet the analysis results of Flow Contracts although, it is one of the priority features to be implemented in the next release, said by an Adacore Senior Engineer.

In Formal Code Verification almost every packages has fully functional properties proved and the un-proved are all from the TR Command List verification. In SPARK, Ghost Code is also considerer for analysis and must be always correct and, therefore, all 13 run-time checks not proved were detected in the Ghost functions used to define the INVARIANT. These warnings are mainly range checks due to the verification of two distinct data-structures compatibility (LIST_G and VALID_POSITIONS (Ghost)). It can be bypassed by adding Pre-Conditions on the INVARIANT function. Still in the Run-time checks, 5 of 119 verifications were proved by Interval, that is, proved by the Ada property of define new ranges for the types and 114 were proved by provers, ones automatic and others with SPARK contracts. 11 Assertions and 2 Functional Contracts not proved, it results from the INVARIANT proof attempt or proof of an INVARIANT specific condition.

As conclusion, the 5% of not proved properties, mostly from TR Command List analysis, do not show how close we were to achieve the full correctness assurance in this component [1]. We can conclude that, at least where it was possible, SPARK analysis works well and that 95% of TR system is proved correct, with high safety and reliable levels. For instance, following many rigorous standards that defends the use of formal methods in pair with software testing, we can affirm that TR implementation (including Command List), according with the tests performed, isl reliable.

In Table 7.2 there is a evaluation about the SPARK performance at each level, see 3.3.3, on this work. Consider for this evaluation the marks: X - Not Good, XX - Good, XXX - Very Good.

[1] An extra effort was performed by the orientation team, with Why3 platform, to reason about the same problem. With the results, we can conclude that the conditions defined for the logical view in this research work are the needed toward the objective.

	STONE	BRONZE	SILVER	GOLD	PLATINUM
TR FDIR	XXX	XX	XXX	XXX	X
TR REGULATION	XXX	XX	XXX	XXX	X
TR COMMAND LIST	XXX	XXX	XX	XX	X
TR MANAGER	XXX	XX	XXX	XXX	X

Table 7.2: SPARK Evaluation.

The SPARK performance at Stone level was Very Good in all components. It was quite effective on the detection of not valid features in SPARK analysis and this obstacles have been overcome in quite effective way too, as for example the implementation of extra Ada package for the use of Access Types avoiding SPARK analysis.

At the Bronze level, in TR Command List package the objects used were all local objects and the use of Flow Contracts to check Data and Flow Dependencies was very efficient. In the others, the evaluation was only Good because it was not possible the Flow Contract check in every subprograms due to his use of Access Types.

Detecting possible run-time errors, Silver level, SPARK performance was Very Good nevertheless, the evaluation on TR Command List was only Good. The target of this level is the detection and the correction of errors, and as presented in Table 7.1 there are some unproved checks from this package.

Achieving Gold level transmits high levels of guarantees that the system will behaviour as expected. For this level, the performance of functional properties proof was Very Good except for TR Command List. It was not possible to prove a set of functional properties required for this package. In general the proof of functional properties had a good performance.

In [27], mention that achieving the Platinum level it is not a easy task and not recommended during initial adoption of SPARK, even harder with SPARK 2014 Discover release without access to advanced SPARK libraries that helps the proof with Lemmas or Predicates. Achieve this level was attempted on TR Command List with the INVARIANT condition, however, without success. Making the evaluation for this level Not Good in all components.

7.4 Recomendations

Based on the extensive analysis of the system code it was possible to identify a set of recommendations to take into account in future space projects in order to guarantee a more optimized code. The recommendations are the following:

- Separation of Concerns, that is, one procedure only must perform one activity (One procedure, one purpose). It will facilitates the validation and verification processes. Thereby, it is only necessary to consider a small set of requirements for each program. For instance, TR Command List package implements the normal data-structure manipulation and the list content update. It must be done in separation;
- Avoid the use of Access Types to interface with data available in other packages. It may result in alias problems which have already caused significant damage in similar systems.

7.5 Conclusion

This chapter presented static analysis performed by SPARK on ExoMars TR software. This analysis was divided in two, Flow Analysis and Formal Code Verification. The first mainly to check the data and flow dependencies and the second to prove the integrity of the code. For the Flow Analysis was demonstrated how have been overcome the use of features not valid for SPARK and the Flow contracts specification for dependencies checks. For Formal Code Verification was presented the techniques used to prove the absence of run-time errors and the software functional properties, with a different demonstration for each component of TR system.

Also was presented the SPARK analysis results and a simple evaluation for the SPARK performance on this work comparing with levels defined in Section 3.3.3. For the Flow Analysis, Stone and Bronze levels, the evaluation is between Good and Very Good, not Very Good just because this analysis was not possible in some components. In Formal Code Verification, Silver, Gold and Platinum levels, the evaluation was more dispersive. For Silver and Gold, the evaluation was also between Good and Very Good with the absence of run-time errors and functional specifications verified in most cases. Although, the Platinum evaluation were Not Good. This level were not achieved at all due to the full functional proof complexity.

With this result, we can conclude that the first four levels of integrity were achieved with very good results. Referring Figure 3.1 on Chapter 3, with this work we gave to ExoMars TR software an integrity level of DAL-A or SIL-4.

Finally, some possible changes to the code implemented and for future implementations were identified. Notations to be consider that will increase the system quality and decrease the difficulty of validation and verification processes.

Chapter 8

Conclusions

In this thesis we have presented the *Design-by-Contract* approach used for a spacecraft on board software formal verification, the ExoMars TGO fully implemented by Critical Software, SA. in cooperation with Thales Alenia Space for the ESA mars exploration mission. This research work results of a collaboration between Critical Software, SA. and University of Beira Interior.

8.1 Results

One difficulty attached to this project was the fact that the analysis was performed in a code already produced without formal verification in mind. First it was necessary to overcome all not valid features for SPARK to proceed with the analysis. Accomplishing that, the next target would be finding bugs in the software that were, possibly, not detected by the extensive and complex verification and validation process. However, finding bugs was not an easy task. We can conclude that was not find any particular bug. The target was first checking the information flow and after assuring the the system safety and functional properties adding an extra confidence to the one already obtained by the verification and validation process performed by Critical Software.

The Flow analysis proves that all data has a correct flow between programs and the absence of variables declaration and initialization problems. The Formal Code Verification prove that the majority of the components achieved the Gold level. That assures the absence of run-time errors and prove the functional properties of the system. With 95% of the annotations proved, it is possible to conclude that the TR system implemented is safety and reliable enough to operate correctly throughout the mission.

To conclude, the attempt to achieve the maximum SPARK integrity level, Platinum, failed. It was not possible to prove the full correctness of the code. However, one should assume that such task is far away from trivial.

8.2 SPARK

SPARK was built with Safety Mission-Critical software verification in mind. Its goal is Ada code verification mostly used for this purpose, Safety Mission-Critical systems development. Due to the mathematical complexity of its annotations it requires high levels of practical knowledge in this area, not being an easy task for beginners. It is recommended an intensive training on this practices before starting to prove really complex systems with it.

The *Design-by-Contract* approach applied by SPARK is based on two analysis, Flow Analysis and Code Verification. It is proved that this approach, besides the complexity, can attach a quality

seal to the produced code. SPARK analysis is not an infallible way to guarantee high reliability but may help towards this goal.

8.3 Critical Software, SA.

For Critical Software, SA. this work results in two very positive aspects. First, there is an increased need on formal methods specialization to introduce this techniques on his development process for a reliable and secure software and, the practice of this work allowed an increased general knowledge acquisition in formal methods with SPARK toolset that can be used in the future Safety Mission-Critical projects in the company, specially for the space exploration. The other important aspect is that, besides the high levels of confidence that the performed tests added to the system and the fact that it has been working very well since system deployment until now, the SPARK analysis came to add a quality seal to the produced software in order of DAL-A, defined by DO-178B which is commonly used in space industries, or SIL-4 defined by IEC 61508.

The company is planning more curricular internships related with the same subject of this research work. Company's goal is to continue the journey started with this project to further increase the knowledge acquired.

8.4 Future Work

The future improvements for this work are the following:

- The initial target for this work were prove correct all spacecraft software systems and the inexperience in this practices only allowed achieve the proof of TR system. For future work is important to achieve the initial target making ExoMars TGO system 100% reliable;
- To get a ExoMars TGO system 100% reliable the Platinum level must be achieved. Not only in TR system but also in GNC, SADM, APM and EDM too. To assume that the system is 100% reliable the system must have all components with full correctness proved.

Bibliography

[1] C. Dross, G. Foliard, T. Jouanny, L. Matias, S. Matthews, J.-M. Mota, Y. Moy, P. Pignard, and R. Soulat, "Climbing the software assurance ladder - practical formal verication for reliable software," in *AVOCS Pre-proceedings*, July 2018. xv, 1, 14

[2] D. Hauzar, C. Marché, and Y. Moy, "Counterexamples from proof failures in the spark program verifier," Ph.D. dissertation, Inria, 2016. xv, 9, 15

[3] C. Loseby, P. Chapin, and C. Brandon, "Use of spark in a resource constrained embedded system," *ACM SIGAda Ada Letters*, vol. 29, no. 3, pp. 87–90, 2009. xv, 19

[4] S. King, J. Hammond, R. Chapman, and A. Pryor, "Is proof more cost-effective than testing?" *Software Engineering, IEEE Transactions on*, vol. 26, pp. 675 – 686, 09 2000. xvii, 22, 23

[5] A. Hall and R. Chapman, "Correctness by construction: Developing a commercial secure system," *IEEE Softw.*, vol. 19, no. 1, pp. 18–25, Jan. 2002. xvii, 23

[6] S. Munari, S. Valle, and T. Vardanega, "Microservice-based agile architectures: An opportunity for specialized niche technologies," in *Reliable Software Technologies – Ada-Europe 2018*, A. Casimiro and P. M. Ferreira, Eds. Cham: Springer International Publishing, 2018, pp. 158–174. 1

[7] L. K. Singh and H. Rajput, "Ensuring safety in design of safety critical computer based systems," *Annals of Nuclear Energy*, vol. 92, pp. 289 – 294, 2016. 1

[8] J. C. Knight, "Safety critical systems: challenges and directions," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, May 2002, pp. 547–550. 1

[9] J. Rushby, "Software verification and system assurance," in *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, Nov 2009, pp. 3–10. 1

[10] E. Conquet, F.-X. Dormoy, I. Dragomir, S. Graf, D. Lesens, P. Nienaltowski, and I. Ober, "Formal model driven engineering for space onboard software," *Proceedings of Embedded Real Time Software and Systems (ERTS2), Toulouse. SAE*, 2012. 1, 19

[11] C. Criteria. Common criteria portal. [Online]. Available: https://www.commoncriteriaportal.org/ 1

[12] B. Beckert, D. Bruns, and S. Grebing, "Mind the gap: Formal verification and the common criteria (discussion paper)," in *VERIFY-2010. 6th International Verification Workshop*, ser. EPiC Series in Computing, M. Aderhold, S. Autexier, and H. Mantel, Eds., vol. 3. EasyChair, 2012, pp. 4–12. 1

[13] W. Wong. Formal methods push toward zero-defect software. [Online]. Available: www.electronicdesign.com/qa-formal-methods-push-toward-zero-defect-software 2

[14] N. Kosmatov, C. Marché, Y. Moy, and J. Signoles, "Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014," in *7th International Symposium on Leveraging Applications*, ser. Lecture Notes in Computer Science, vol. 9952. Corfu, Greece: Springer, Oct. 2016, pp. 461–478. 2

[15] J. W. McCormick and P. C. Chapin, *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015. 2

[16] J. Barnes, *Programming in Ada 2012*. Cambridge University Press, 2014. 2

[17] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich, "Why3 Shepherd your herd of provers," in *Boogie 2011 First International Workshop on Intermediate Verification Languages*, 2011, pp. 53–64. 2, 15

[18] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *Software Engineering and Formal Methods*, G. Eleftherakis, M. Hinchey, and M. Holcombe, Eds. Springer Berlin Heidelberg, 2012. 2

[19] O. Corporation. Java. [Online]. Available: https://www.java.com/ 2

[20] P. Baudin, J. christophe Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto, and I. S. Île-de france, "Acsl: Ansi/iso c specification language," 2008. 2

[21] G. Leavens and Y. Cheon, "Design by Contract with JML," *Draft, available from jml-specs.org*, 2006. 2

[22] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, Oct 1992. 5

[23] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969. 5

[24] D. Hauzar, C. Marché, and Y. Moy, "Counterexamples from proof failures in spark," in *International Conference on Software Engineering and Formal Methods*. Springer, 2016, pp. 215–233. 9, 15

[25] R. Chapman and F. Schanda, "Are we there yet? 20 years of industrial theorem proving with spark," in *International Conference on Interactive Theorem Proving*. Springer, 2014, pp. 17–26. 9, 22, 25

[26] C. Dross and Y. Moy, "Abstract software specifications and automatic proof of refinement," in *International Conference on Reliability, Safety, and Security of Railway Systems*. Springer, 2016, pp. 215–230. 13

[27] T. AdaCore, "Implementation guidance for the adoption of spark," 2018. 14, 39, 52

[28] J. Guitton, J. Kanig, and Y. Moy, "Why hi-lite ada?" in *Boogie 2011 First International Workshop on Intermediate Language Verification*. Citeseer, 2011, p. 27. 14, 15

[29] J. Kanig, E. Schonberg, and C. Dross, "Hi-lite: The convergence of compiler technology and program verification," in *Proceedings of the 2012 ACM Conference on High Integrity Language Technology*. New York, NY, USA: ACM, 2012, pp. 27–34. 14, 20

[30] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout, "The alt-ergo automated theorem prover," *http://alt-ergo.lri.fr*, 2008. 15

[31] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 171–177. 15

[32] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340. 15

[33] G. Le Lann, "The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems apport," p. 33, 1996. [Online]. Available: https://hal.inria.fr/inria-00073613/ 17

[34] T. Tolker-Nielsen, "EXOMARS 2016 - Schiaparelli Anomaly Inquiry," *Paris, France2017*, 2017. [Online]. Available: http://exploration.esa.int/mars/59176-exomars-2016-schiaparelli-anomaly-inquiry/ 17

[35] C. Brandon and P. Chapin, "A spark/ada cubesat control program," in *International Conference on Reliable Software Technologies*. Springer, 2013, pp. 51–64. 17

[36] W. D. Lakin and C. Brandon, "Landing a cubesat payload on the moon: the vermont space grant lunar lander project," *Design Principles and Practices: An International Journal*, vol. 5, no. 3, pp. 79–88, 2011. 17

[37] C. Brandon, "A navigation test flight for a lunar cubesat," in *SpaceOps 2012 Conference*. American Institute of Aeronautics and Astronautics, Inc., 2012. 17

[38] C. Brandon and T. Cockram, "Use of ada in a student cubesat project," *Ada User Journal*, vol. 29, no. 3, p. 213, 2008. 17

[39] C. Brandon and P. Chapin, "High integrity software for cubesats and other space missions," 10 2015. 18

[40] ——, "The use of spark in a complex spacecraft," *ACM SIGAda Ada Letters*, vol. 36, no. 2, pp. 18–21, 2017. 18

[41] D. Lesens, Y. Moy, and J. Kanig, "Formal Validation of Aerospace Software," in *DASIA 2013 - DAta Systems In Aerospace*, ser. ESA Special Publication, vol. 720, Aug. 2013, p. 43. 19

[42] C. Dross, P. Efstathopoulos, D. Lesens, D. Mentré, and Y. Moy, "Rail, space, security: Three case studies for spark 2014," *Proc. ERTS*, 2014. 19

[43] H.-L. partners, "L7.2.2 deliverable - final report on experimentations on industrial case studies," http://www.open-do.org/wp-content/uploads/2013/05/Industrial_Case_Studies_Final_Report.pdf, 2013. 21

[44] M. Croxford and R. Chapman, "Correctness by construction: A manifesto for high-integrity software," 2005. 22, 26

[45] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, no. 4, pp. 19:1–19:36, Oct. 2009. 22

[46] E. Brito, "A (Very) Short Introduction to SPARK: Language, Toolset, Projects, Formal Methods & Certification," *INForum*, pp. 479–490, 2010. 22

[47] J. M. Spivey and J. Abrial, *The Z notation*. Prentice Hall Hemel Hempstead, 1992. 22

[48] I. Toyn and J. A. McDermid, "Cadi: An architecture for z tools and its implementation," *Software: Practice and Experience*, vol. 25, no. 3, pp. 305–330, 1995. 22

[49] "Multos," https://www.multos.com/, accessed: 2019-04-15. 23

[50] D. L. Haven, "The newest seamless airlifter: The c-130j-30." AIR FORCE INST OF TECH WRIGHT-PATTERSONAFB OH SCHOOL OF LOGISTICS AND ACQUISITION MANAGEMENT, Tech. Rep., 1998. 24

[51] M. Croxford and J. Sutton, "Breaking through the v and v bottleneck," in *Ada in Europe*, M. Toussaint, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 344–354. 24

[52] S. R. Faulk, L. Finneran, J. Kirby, and A. Moini, "Consortium requirements engineering guidebook," Technical Report SPC-92060-CMC, Software Productivity Consortium, 2214 Rock …, Tech. Rep., 1993. 24

[53] S. Faulk, L. Finneran, J. Kirby, S. Shah, and J. Sutton, "Experience applying the core method to the lockheed c-130j software requirements," in *Proceedings of COMPASS'94 - 1994 IEEE 9th Annual Conference on Computer Assurance*, June 1994, pp. 3–8. 24

[54] "National security agency | central security service," https://www.nsa.gov/, accessed: 2019-01-13. 24

[55] D. Cooper and J. Barnes, "Tokeneer ID Station: EAL5 Demonstrator: Summary Report," *Praxis*, pp. 1–73, 2008. 24

[56] Y. Moy and A. Wallenburg, "Tokeneer: Beyond formal program verification," *Embedded Real Time Software and Systems*, 2010. 24

[57] J. Barnes, R. Johnson, and J. Widmaier, "Engineering the Tokeneer Enclave Protection Software," *International Symposium on Secure Software Engineering (ISSSE'06)*, March 2006. 24

[58] "Nats." [Online]. Available: https://www.nats.aero/ 25

[59] F. Gouvernment, "Agence nationale de la sécurité des systèmes d'information." [Online]. Available: https://www.ssi.gouv.fr 25

[60] R. Benadjila, M. Renard, P. Trebuchet, P. Thierry, A. Michelizza, and J. Lefaure, "Wookey usb devices strike back," in *Symposium sur la sécurité des technologies de l'information et des communications*, 2018. 25

[61] AdaCore. Adacore. [Online]. Available: https://www.adacore.com 26

[62] P. Messina, D. Vennemann, and B. Gardini, "The european space agency exploration programme aurora," in *1st Space Exploration Conference: Continuing the Voyage of Discovery*, p. 2518. 27

[63] D. Bonetti, G. D. Zaiacomo, G. Blanco, I. P. Fuentes, S. Portigliotti, O. Bayle, and L. Lorenzoni, "Exomars 2016: Schiaparelli coasting, entry and descent post flight mission analysis," *Acta Astronautica*, vol. 149, pp. 93 – 105, 2018. 27

[64] C. Donlon, B. Berruti, A. Buongiorno, M. Ferraria, J. Frerick, P. Goryl, U. Klein, H. Laur, C. Mavrocordatos, J. Nieke *et al.*, "The global monitoring for environment and security (gmes) sentinel-3 mission," *PROCEEDINGS OF THE GHRSST XII SCIENCE TEAM*, 2011. 29

[65] P. Largeteau, "Ostrales/sparc: an ada real-time kernel for on-board satellite software application," *EUROPEAN SPACE AGENCY-PUBLICATIONS-ESA SP*, vol. 447, pp. 299–308, 1999. 29

[66] M. Casasco, B. Girouart, and A. Benoit, "Gnc and aocs functional chains engineering and verification," *8th ESA Workshop on Avionics, Data, Control and Software Systems - ADCSS 2014*, 2014. 37

[67] P. Neto, "Exomars tgo tr system code base verified with spark," 2019. 41, 42, 44, 47, 49

[68] R. Cauderlier and M. Sighireanu, "A verified implementation of the bounded list container," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10805 LNCS, pp. 172–189, 2018. 45