Universidade do Minho
Escola de Engenharia

Daniel Ribeiro Quinta

Application of Formal Methods in the ITASAT project

Braga, July 2013

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Daniel Ribeiro Quinta

Application of Formal Methods in the ITASAT project

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob a orientação de:
Professor José Creissac Campos (UMinho)
Professora Emilia Villani (ITA)

Braga, July 2013

Acknowledgements

Firstly, I would like to thank Professor José Creissac Campos, for all the help, support and encouragement throughout the research and writing of this dissertation. Thank you for being a dedicated mentor and guide, also considering that the major part of the work has been made at distance. A special thanks also to Professor Emilia Villani for the hosting and guidance provided at ITA (Instituto Tecnológico de Aeronautica) and INPE (Instituto Nacional de Pesquisas Espaciais), which made this research possible.

Secondly, I thank my mother, for her endless love and support in every moment of my life.

I thank Elena for her affection and patience. Part of this thesis is yours.

I also want to thank my friends and colleagues, especially André, Iago, Pedro e Rui, for the help and advice given as much as a good laugh.

I would also like to mention my cousins, not for their help in concentrating, but for their endless cheerfulness.

Abstract

Critical software can be potentially dangerous if not well verified, leading to serious failures. Accordingly, there is a need for improved validation and verification methods in order to have guarantees about the software final product. The aim of this project is to define a more linear and organized verification and validation plan to, formally, verify the most critical parts of the OBDH (On-Board Data Handling) subsystem of ITASAT, supported by the Alloy formal language.

Alloy supports the description of systems whose state involves complex relational structure. The application of Alloy and Alloy Analyzer was motivated by the need for a formal specification that is more closely tailored to state-machines, and more amenable to automatic analysis. Structural and behavioural properties are described declaratively, by conjoining relations and constrains, making it possible to develop and analyze a model incrementally. Due to the high cost of using these methods, they are mainly used in the development of high-critical software where safety and security are crucial.

This dissertation presents a set of guidelines for analysis and modelling of software systems which support the creation of a formal model and allow some extra behaviours such as synchronization, interruptions and flags. A new tool, ModelMaker, was developed in order to create models using these guidelines in a more interactive way.

Resumo

O *software* crítico torna-se potencialmente perigoso quando não cuidadosamente verificado, podendo resultar em falhas graves. Por consequência, existe uma necessidade de melhorar os métodos de validação e verificação, oferecendo garantias sobre o produto final.

O objetivo do presente projeto é a definição de um plano de verificação e validação linear e organizado para verificar formalmente as partes mais críticas do sub-sistema OBDH (On-Board Data Handling) do Satélite do ITA (Instituto de Tecnologia Aeronáutica), através da linguagem formal Alloy.

Alloy permite a descrição de sistemas cujo estado envolve estruturas relacionais complexas. A sua aplicação e a do Alloy Analyzer justificou-se pela necessidade de uma especificação formal que fosse mais adapta a máquinas de estado e mais recetiva a análises automáticas. As caraterísticas estruturais e comportamentais são descritas declarativamente, através da junção de relações e restrições, permitindo desenvolver e analisar um modelo de forma iterativa. Devido ao seu custo elevado, estes métodos são principalmente usados no desenvolvimento de *softwares* críticos, para os quais a segurança é um pressuposto fundamental.

Esta dissertação apresenta um conjunto de diretrizes para a análise e modelação de sistemas. Estas suportam a criação de um modelo formal e permitem alguns comportamentos adicionais como a sincronização, a interrupção e *flags*. É também introduzido o *ModelMaker*, uma ferramenta que, fazendo uso das directrizes, ajuda numa construção interativa e automática de modelos.

Para a minha Mãe.

Contents

Acknowledgements **v**

Abstract **vii**

Resumo **ix**

1 Introduction **1**
1.1 Motivation 1
1.2 Problem 2
1.3 Objective 3
1.4 Contributions 5
1.5 Structure of the Document 5

2 ITASAT **7**
2.1 Introduction 7
2.2 ITASAT 8
2.3 Architecture 11
2.4 Attitude Control and Data Handling 12
2.5 Conclusion 13

3 V & V in Critical Systems **15**
3.1 Introduction 15
3.2 ESA Document 17
3.3 NASA Document 18
3.3.1 NASA Guidebook I 18
3.3.2 NASA Guidebook II 19
3.3.3 Levels of Formalisation 19
3.4 Formal Approach 20
3.4.1 Verification Methodology 20
3.4.2 Formal Approach 21
3.5 Conclusion 22

4 Alloy **23**
4.1 Introduction . . . 23
4.2 Atoms and Relations . . . 24
4.3 Alloy Logic . . . 26
4.3.1 Propositional Logic . . . 27
4.3.2 First-Order Logic . . . 28
4.4 Alloy Language . . . 28
4.4.1 Operators and Quantifiers . . . 29
4.4.2 Functions, Facts, Assertions and Predicates . . . 30
4.4.3 Run and Check . . . 33
4.5 Alloy Analysis and Analyzer . . . 34
4.5.1 Alloy Analysis . . . 35
4.5.2 Alloy Analyzer . . . 36
4.6 Conclusion . . . 36

5 Formal Model Template **37**
5.1 Introduction . . . 37
5.2 Phase 1: Formal Model Template . . . 38
5.2.1 Initial Model . . . 38
5.2.2 Requirements and Facts . . . 41
5.2.3 Move your Model . . . 42
5.2.4 Synchronization . . . 44
5.2.5 System Exceptions . . . 47
5.2.6 Booleans, Integers and Flags . . . 48
5.3 Phase 2: ModelMaker Automatic Building . . . 51
5.3.1 Create a Formal Model . . . 52
5.3.2 Checking the Model . . . 54
5.3.3 ModelMaker Implementation . . . 56
5.4 Conclusion . . . 56

6 A Real Case Study! **59**
6.1 Introduction . . . 59
6.2 Satellite System Modules . . . 59
6.2.1 Unreachable Survival Mode . . . 60
6.2.2 Starvation in Alignment Payload Mode . . . 61
6.2.3 Duality in Alignment Payload Mode . . . 61
6.2.4 Payload Mode Smooth Exit . . . 62
6.3 Software Modules . . . 64
6.3.1 Unreachable CM Initialization . . . 64
6.3.2 Incompatibilities in Communication . . . 66
6.3.3 OBC Sequential Output . . . 66
6.3.4 OBC Nominal Smooth Exit . . . 67
6.3.5 Inconsistency between Diagrams and Requirements . . . 67

6.4 Communication Modules - Receiver 68
6.4.1 Foreground Routine - Receiver 70
6.4.2 Survival Flag Interruption 70
6.4.3 Memory Read and Write Pointers 72
6.5 Conclusion 74

7 Conclusion and Future Work **77**
7.1 Conclusion 77
7.2 Future Work 78

List of Figures

1.1 Verification and Validation Model, adaptation from 2
1.2 The life-cycle of Validating and Verifying the requirements using Alloy. . . 4

2.1 Mission Concept, adaptation from Larson/Wertz, Space Mission Analysis and Design Workshop [2012]. 10
2.2 Diagram of the ITASAT satellite system 12
2.3 OBC Hardware Diagram 13

4.1 Conjunction of facts ($list <= 4 \wedge list < 3$) 32
4.2 Checking assertions with Alloy `Check` command 35

5.1 Initial model with events 39
5.2 An example of dynamic modelling simulation in Alloy model. 45
5.3 Action Synchronization in model A and model B 47
5.4 Main model and Exception model 49
5.5 "A Satellite cannot be on earth and in space at the same time", Bool example 50
5.6 "Counter" incrementation throw Instant 52
5.7 ModelMaker Interface 53
5.8 State and Event addition to the model in ModelMaker 53
5.9 A new model creation in ModelMaker 54
5.10 Alloy file resulting as a ModelMaker output 55
5.11 ModelMaker model created by ModelMaker tool 56

6.1 Satellite modules ITA documentation 60
6.2 Simplification in Satellite System Modules ITA 62
6.3 Alloy Analyzer found a invalid `Instant` that violates the invariant(one `CurrentState`) 63
6.4 System modules ITA documentation 65
6.5 Software modules alloy 67
6.6 Overlap of Software Modules by Satellite Modules 69
6.7 Receiver, Communication with OBC diagram [ITASAT documentation] . . 70
6.8 Communication with OBC, Interruption Alloy Model 72
6.9 Communication with OBC, Interruption Alloy Model 73
6.10 Three different states (1,2,3) of the Memory 75

List of Tables

3.1 Mechanical Support for Specification and Analysis Phases of FM 16
3.2 Levels of Formalisation and Scope of Formal Methods Used 20

4.1 Multiplicities in a Binary Relation 25
4.2 Different Types of a Binary Relation, adaptation from FM Classes 25
4.3 Different Logic Styles 27
4.4 Alloy Quantifiers 30

6.1 Alignment between Satellite Modules and Software Modules 68

Chapter 1

Introduction

1.1 Motivation

One of the rewarding and challenging tasks in software development is to know that the systems that are being designed and constructed are high quality products that people can rely on. This becomes more and more important when we talk about embedded critical environments such as transportation networks, aeronautic and airspace control, weapons deployment and activation systems, international banking transactions adjustments, controlled chemical and physical environments (e.g., nuclear power plant and particle accelerator), among others.

Developers of this type of software have to be sure that there is no possibility of either errors and failures that may cause tragical accidents with human lives being at risk, or bad management of significant amounts of money due to software malfunction. In an effort to find and correct these possible errors, building a perfect software piece (bug free), teams try to document and design a full test strategy, side-by-side with the software evolution plan. This is a possible approach on how the two sides can work side-by-side. Figure 1.1 shows that for every important step made in the development design, there should be a correspondent verification and validation testing process monitoring and following those steps.

Software validation and verification activities check the software against its own requirements and specifications. In order to do this there are some guidelines that should be followed. Furthermore, every project must be verified and validated, as far as possible, by someone other than the author.
The steps of the verification progress are:

- checking that every unit meets specified requirements;
- ensuring that the process of including an item is safe;
- the effort on verifying and validating an item should be adjusted to the critical level that will be operationally used.

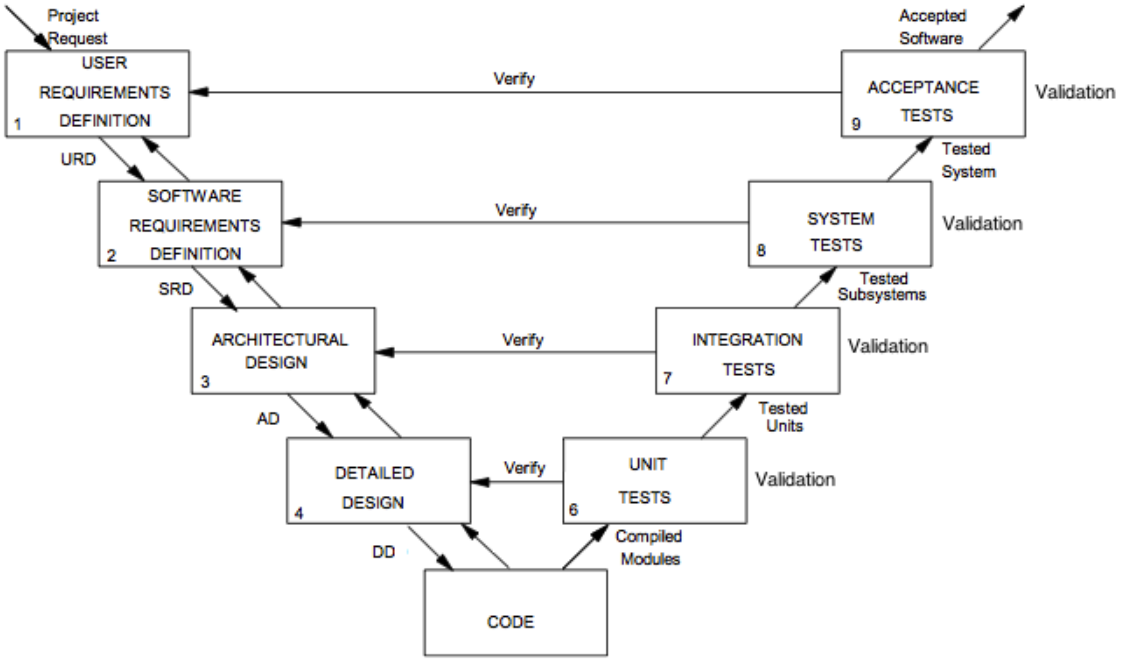

Figure 1.1: Verification and Validation Model, adaptation from Jones and Mortensen [1995].

1.2 Problem

Software verification and validation is very important and affects directly the quality of the software; the probability of making errors is higher especially in big projects and where the faults can be propagated easily. Errors that are left behind become more and more hard to find and correct. People are not perfect and do mistakes: a software that has not been verified will increase strongly the probability of operational failure. The objective of this attestation is to reduce software errors to an acceptable level. The effort needed depends upon how critical and how complex the software is [Beizer and Reinhold, 1983].

Further, it has been observed that organizations using just traditional testing methods are restricted to a certain level of quality in terms of safety. This means that a point is reached where major improvements are no longer possible due to the technology used. Even with exhaustive tests on the software produced, these organizations still have some faults and defects they cannot control [Kelly et al., 1992]. In order to fight against them, formal methods techniques were introduced in the project life-cycle. Formal methods are logical (mathematical) techniques, often supported by tools, for developing and analysing software and hardware systems. Mathematical rigour enables users to analyse and verify these models at each of the development phases: requirements engineering, specification, architecture, design, implementation, testing, maintenance, and evolution.

Being aware that there are many issues to be taken into account when considering incorporating formal methods in software development is very important. Many levels of accuracy, as well as a large variety of formal methods depending on the phase of software development, can be adopted. The cost/benefits analysis of the use of formal methods is in fact an important issue, with project management factors as well as technical factors to be considered.

The Brazilian Institute of Aeronautics Technologies (ITA) is building a Satellite, the ITASAT project. In such a complex project in terms of human resources, money invested and financing, a good planning is critical to reduce the costs and to improve the project quality. In order to avoid changes in the project in a later phase, verification and validation must be performed. In fact, those changes might imply extra costs and cause the project to go overbudget, due to the extra labour hours or space equipment changes, that in this situation could mean paying extra thousands for a new processor or for a communication antenna.

1.3 Objective

To address this issues and to improve and facilitate the usage of formal methods in verification and validation of a critical project design, this thesis proposes the creation of a set of guidelines in order to help the user to build a model in a more easy and smooth way. This provides the project manager with an initial overview of the project in a very short period of time, allowing to model the early project phase as well as the more complex final approach. The model can grow as the project grows and gives feedbacks in useful time.

The main focus is to assist the ITASAT work-group in the creation and development of such a critical software aerospace project. We want to define a more linear and organized verification and validation plan to, formally, verify the most critical parts of the project, supported by Alloy models and proceed side-by-side with a formal modelling in several of the project different phases. The work follows a direct line from Architectural Design, Communication Module to Transceiver Code.

The goal of this project, as shown in Figure 1.2, is to extract the requirements from the documentation, create an Alloy Model that enables formal verification and validate those models again with the former documentation. To help the Alloy Model creation, an automatic model maker tool would be very useful.

We will focus our efforts in the creation of the models and analysing those models in order to find faults with a formal model checker. In this thesis, we assess the current state of the art in Verification and Validation (VV) and the application of Formal Methods (FM) in the software development process, concentrating on their increasing use at the earlier stages of specification and design in critical software projects.

The objective is to study the application of Formal Methods in the development process of ITASAT (satellite of Aeronautical Institute of Technology - ITA). The OBDH (On-Board Data Handling) subsystem will be used as a case study. The work will be divided into three phases: in the first part, guidelines for a formal model are going to be developed with some

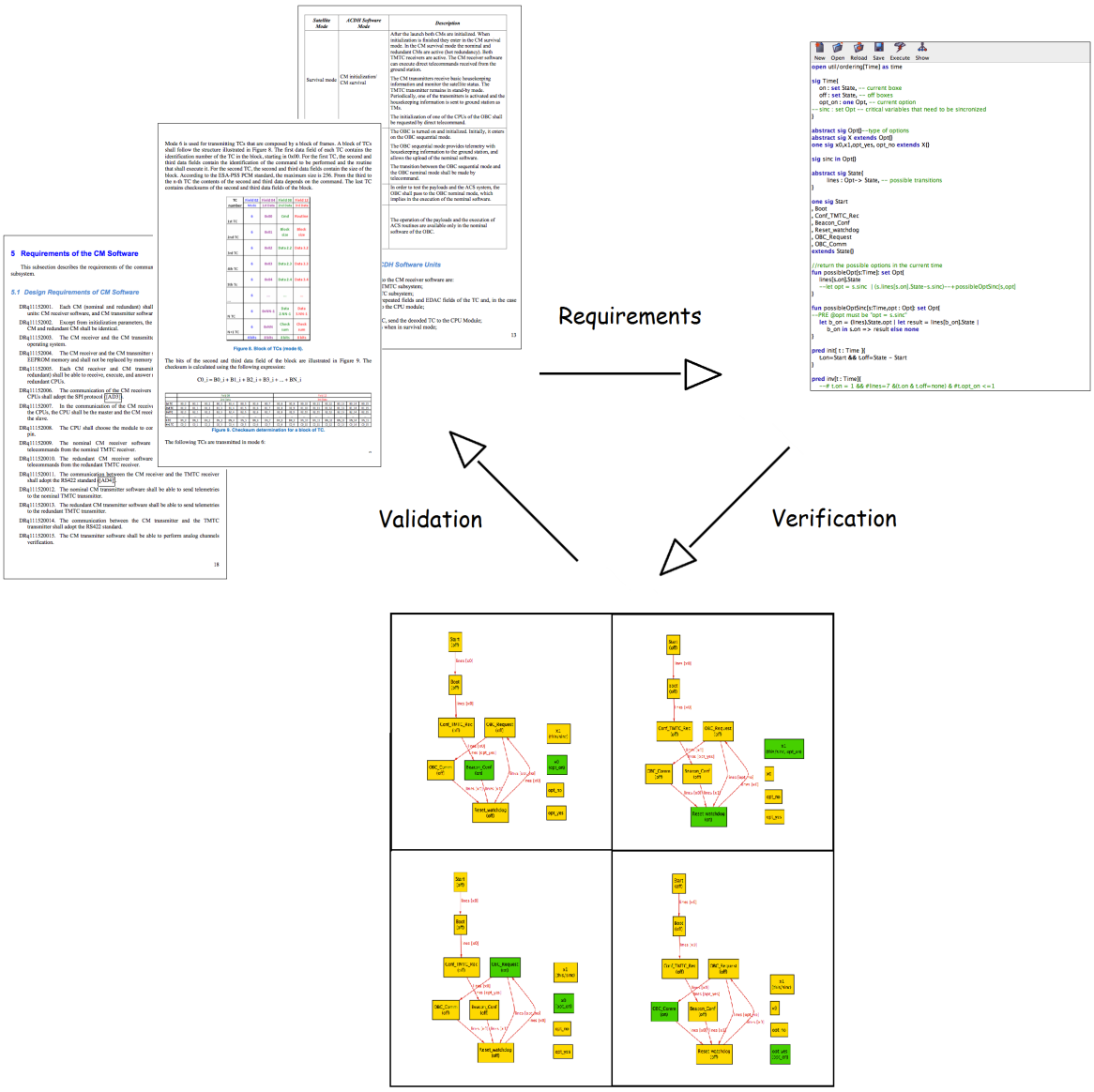

Figure 1.2: The life-cycle of Validating and Verifying the requirements using Alloy.

extra features, as a support for the creation of a more reliable model to be used with the Verification and Validation methods; then, in second phase, a tool will be created to support the model creation, making use of the pre-stablish rules in the guidelines in phase 1. Finally, in the third phase, a Formal Methods approach with model verification will be made in the different project phases to give a mathematical formalism support in the specification, development and verification of software system, trying to contribute to the reliability and robustness of the design, performing appropriate mathematical analysis.

1.4 Contributions

Due to the complexity of the entire project, that joins together several engineering areas - mechanics, electronics and informatics - the work was based mainly on the documentation. It was not possible to make use of some formal tools, which are used for static code analysis (i.e. BLAST), in the code creation phase owing to deep project alterations at the mechanical and structural level. However, a system architecture model methodology was developed to verify and validate some critical modules against their specifications, restrictions and requirements. This guidelines can thus be used to help increasing confidence in the correctness of a specification and the related implemented or modified code, and to make it easier to keep the specification and code in sync.

The structural methodology guidelines were defined to help creating the models from the documentation, using the Alloy language and using those models for validating the specification.

Alloy Analyzer proved to be useful to automatically find flaws in the models and help in the verification process of the project's critical requirements. The ModelMaker tool was designed and created to support in the model creation process. It proved to be very helpful and time saving to build and to initiate the model. Some features were added, such as: cycles verification, limited transition access and checking starvation. This tool makes the process of building and starting checking the requirements much more easy and productive.

Thanks to this initial project quick modelling, some errors and flaws were found, and some logistic incompatibilities in the communications modules that could derive into high-cost mechanical modifications were detected, thereby allowing a correction in the embryonic stage of the project.

Although those guidelines were elaborated and directed for a satellite modelling, they can be used to model other systems, to simulate a dynamic modelling and to automatically check the requirements and the overall project design.

1.5 Structure of the Document

The document is structured as follows: Chapter 2 the work that has been made at ITASAT is presented; then, in Chapter 3, we discuss the reasons for testing critical software, showing the effort of two important international aeronautic agencies, the European Space Agency (ESA) and the National Aeronautics and Space Administration (NASA), the first in VV and the other in FM, finalizing with a brief introduction to FM tools and methodologies; in Chapter 4 Alloy language and the Alloy Analyzer tool is briefly described. Chapter 5 ,"Using Alloy", is divided into two parts: the first describes our guidelines for building an Alloy initial model with extra features, and in the second part the new tool ModelMaker is presented and succinctly described. In Chapter 6, the application of both of the guidelines and of the created tool in some phases of the ITASAT project, is illustrated with several examples. Finally, in Chapter 7, we draw some conclusions about current practice and we outline future work.

Chapter 2

ITASAT

2.1 Introduction

A satellite or human satellite is an object which has been placed into orbit by humans. Several countries build satellites, but just a few have the ability to launch them. Brazil has a launching zone located in Alcântara Launch Center (Centro de Lançamento de Alcântara– CLA), in Maranhão. There are several launching centers in the world but this one has an increased importance due to some special characteristics:

Equator Line Proximity Saves fuel and helps the launchers's momentum when exiting the Earth;

Vertical and Horizontal Earth Launching It is possible to launch satellites to equatorial orbits and polar orbits.

Low Demographic Density With such a big center area is possible to have several silos for different rockets.

Stable Clime Raining season well defined and low winds rates make launching a satellite possible almost all year.

With a such special location and characteristics, the Alcântara Launch Center became an important piece in the airspace strategy and industry.

The ITASAT project, the first Brazilian university satellite, is a part of the Multi-Year Development Plan and Launching Small Technological Satellites, aimed at promoting and supporting the Brazilian demand for future generations of micro and nano-satellites. The overall coordination of the project is made by the Brazilian Space Agency (AEB), while the Aeronautics Technological Institute (ITA) is responsible for the project execution and the National Institute for Space Research (INPE) provides technical advise, laboratory infrastructure and financial management. ITASAT was designed to work as a weather satellite. There are several applications for space satellites, which, among other functions, are used for:

Navigation, Tracking and Mapping The use of Global Position System (GPS), for helping in vehicles routes, such as car, boats, planes, etc. Tracking devices are implanted in all kind of objects such mobile phones, cars, black boxes, etc. Thanks to satellites it is possible to access a global map of the Earth.

Communication Satellites can be used for controlling direct communications instead of using the Earth structures.

Military and Security Satellites are constantly scanning and keeping observation on hostile territories around the globe, providing images and data to military command and intelligence agencies.

Science and Discovery Cameras and other scientific instruments can go through space and return information to ground stations with important informations in order to know more about the planets in our solar system and the space itself.

Entertainment Television and radio are nowadays supported by satellites organizations.

Meteorology and Geology Satellites also enable us to scan the surface of the Earth for geological research and geological analysis, particularly to predict when an active volcano is about to launch or spot the early warning signs of a developing hurricane. They can monitor atmospheric changes in the Earth's atmosphere, enabling us to predict and forecast the weather.

Satellites are still very expensive and really difficult to maintain. Abandoned satellites contribute to grow the space debris, making it dangerous for space travels and increasing the danger of damages in shuttles when they try to leave and enter the atmosphere.

The Space Surveillance Network (SSN) has tracked a total of more than 24,500 objects in space, about 20,000 pieces of "space junk" in low-Earth orbit, most of which are discarded bits of spacecraft or debris from collisions in orbit. SSN is now watching about 8,000 objects currently in orbit. A few hundred satellites are currently operational, whereas thousands of unused satellites and satellite fragments orbit the Earth as space debris. The United States Strategic Command is primarily interested in the active satellites, but also tracks space debris which upon re-entry might otherwise be mistaken for incoming missiles.

2.2 ITASAT

With an annual budget of $1.7 million, ITASAT is a micro-satellite for collecting environmental and weather data. This small satellite was designed to be launched as a "piggyback" payload of an primary satellite mission. This options allows the secondary satellite to travel in the same structure, since unification among payloads and launchers enables quick exchanges of payloads and utilization of launch opportunities on short notice.

The ITASAT project is a combination of several areas: communication, electronic, mechanic, computer engineering and science, all of them aiming at airspace projects. The work group is composed approximately of twenty Master degree students, and supervisors from INPE and ITA to provide some orientations. It is a high complexity project, since it involves developing a software considering the restrictions of all the project related areas: periodic communication, space reduction, expensive materials, short memory and slow processors, etc.

It is interesting to note that, in the aeronautic and airspace systems, every item included in the system is extremely important and has a vital function in the project. Otherwise, it would not be included due its high cost. Figure 2.1 shows what is involved when planning a satellite mission, and how complex it is. The system is divided in:

Subject The objective of the experience. The objective of ITASAT mission is getting information about the weather.

Orbit and Constellations When sending a satellite to the space it is necessary to know its trajectory and its role in the organization of the other satellites.

Payload, Space Element, Spacecraft Bus The satellite itself with the instruments to get the information and studies from the space.

Launch Element In order to launch a satellite to space is necessary a transporter. In this case, ITASAT is a piggyback satellite and it will use a transporter from a primary satellite.

Ground Element The infrastructure on earth that allows the communication between Earth station and the satellite in the space.

Mission Operation The "intelligence" on Earth is a group of people that is responsible for taking decisions and reacting to the given satellite informations.

Command, Control and Communications Architecture It is vital to plan the system architecture. That involves all the actions-reactions decisions in communication and define the procedures.

The national investment in the space area, provides a series of missions designed to conduct experiments, develop and test innovative technology of satellites and payloads, and enables the Brazilian space industry in this segment. The first phase of the program was created in 2003 with the participation of the ITA and INPE, to study possible ways of University - Research Institutes - Industry - Government, interaction in implementing such a program, and to study the main technological aspects involved in conducting a mission in the program: satellite subsystems, integration and testing, launch, ground segment, operation, management and project documentation. Thus the Project "ITASAT-1 Mission" was the first project-mission of this program.

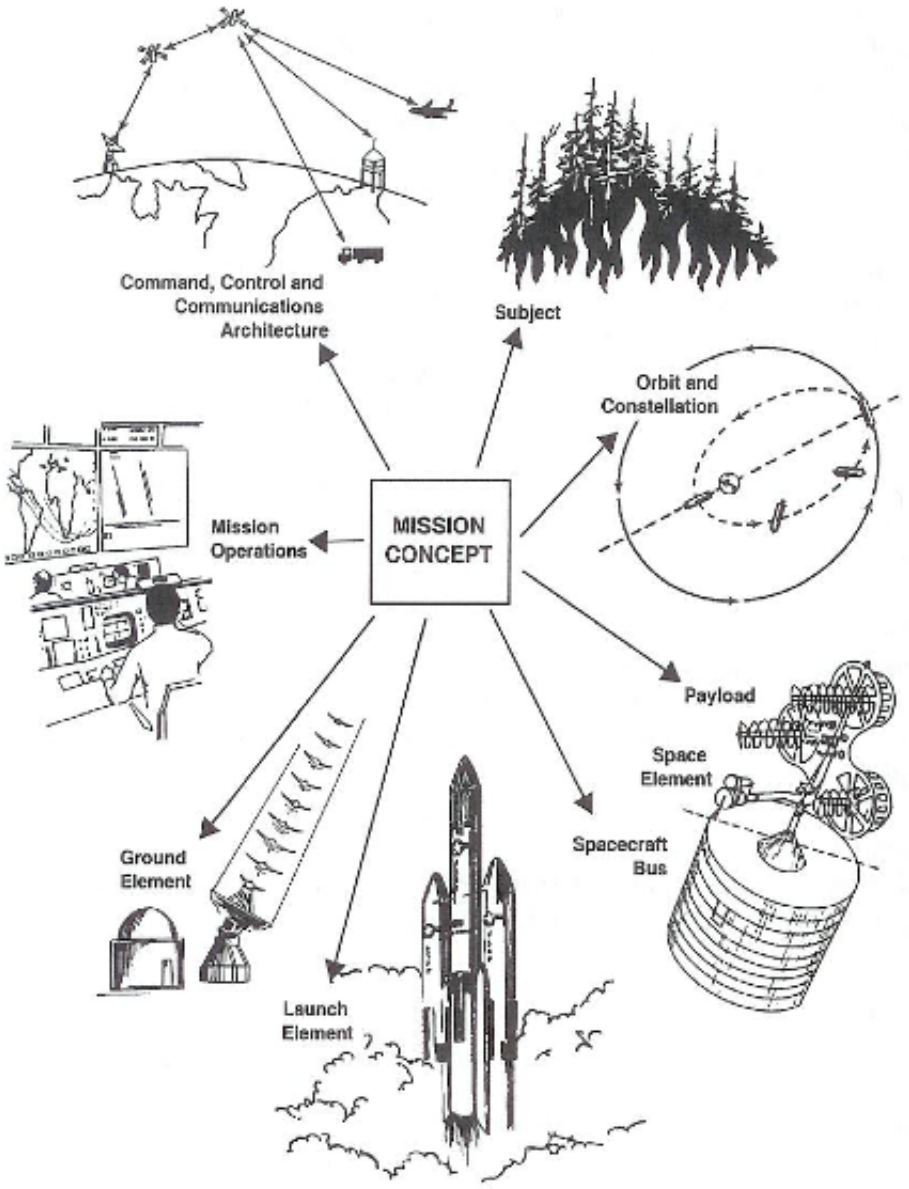

Figure 2.1: Mission Concept, adaptation from Larson/Wertz, Space Mission Analysis and Design Workshop [2012].

Given the novelty of the initiative and its institutional coverage, the project went through several stages before being consolidated. After its creation, various forms of collaboration and participation of the institutions involved were studied. The type of satellite and mission that would be up to an university satellite was also discussed and ultimately there was an attempt to study various aspects of the development of small satellites (for micro-satellites weighing less than 100kg).

2.3 Architecture

As Figure 2.2 shows, the satellite mechanical and structural subsystem (MSS) is divided in four main modules: attitude control and data handling (ACDH), electric power (EPS), telemetry and telecommand (TMTC) and the satellite payloads.

ACDH performs on board data handling and attitude control routines. To perform the satellite attitude control, it contains sensors and actuators. The sensors are three solar sensors and a three axis magnetometer that can be used for experimental attitude estimation and control algorithm. The actuators are three air coils. The ACDH interacts with the TMTC to manage the communications, receiving and controlling the energy level in the EPS, and ACDH manages the payloads operation.

TMTC subsystem has two transceivers (nominal and redundant). Each transceiver is composed of a receiver, that receives telecommands (TC) from the ground station, and a transmitter, that transmits telemetrics (TM) to the ground station. The receivers are configured in hot-standby redundancy (always listening), and the transmitters are configured in cold-standby redundancy (just activated when needed). TMTC module communicates with the main module due to communication modules (CMs) in ACDH.

EPS can store and accumulate energy due to its solar panels. tIs objective is to support the power demands and to warn the main module if the battery reaches a certain low point.

PAYLOAD The satellite payload has four experiments: the digital data collection system (DCS), an acetone-aluminium heat pipe, an attitude estimator equipment with memories (MEMS), and the inter satellite link equipment (FoX/ISL).

Every module is a vital module, otherwise it would not be part of the satellite. Without the TMTC module, neither the collected data management, nor the reception of commands from the ground station would be possible. Without the EPS, the satellite would not have energy to survive in space. If the payloads cargo failed, the satellite would turn into a simple metal box sending a beep with unnecessary information from the space. If the main control module ACDH was lost, there could not be communication or the information received could not be managed, resulting in a full functional "body" machine without a "brain".

Student teams have to analyse the orbit properties for ground station visibility and related implications on the different satellite subsystems, such as on board data handling, power, telecommunications, attitude determination and control, thermal control and structure.

The selected area in the ITASAT satellite system diagram in Figure 2.2 represents the communication between the On-Board Computer and the communication module and it is an important part of our study.

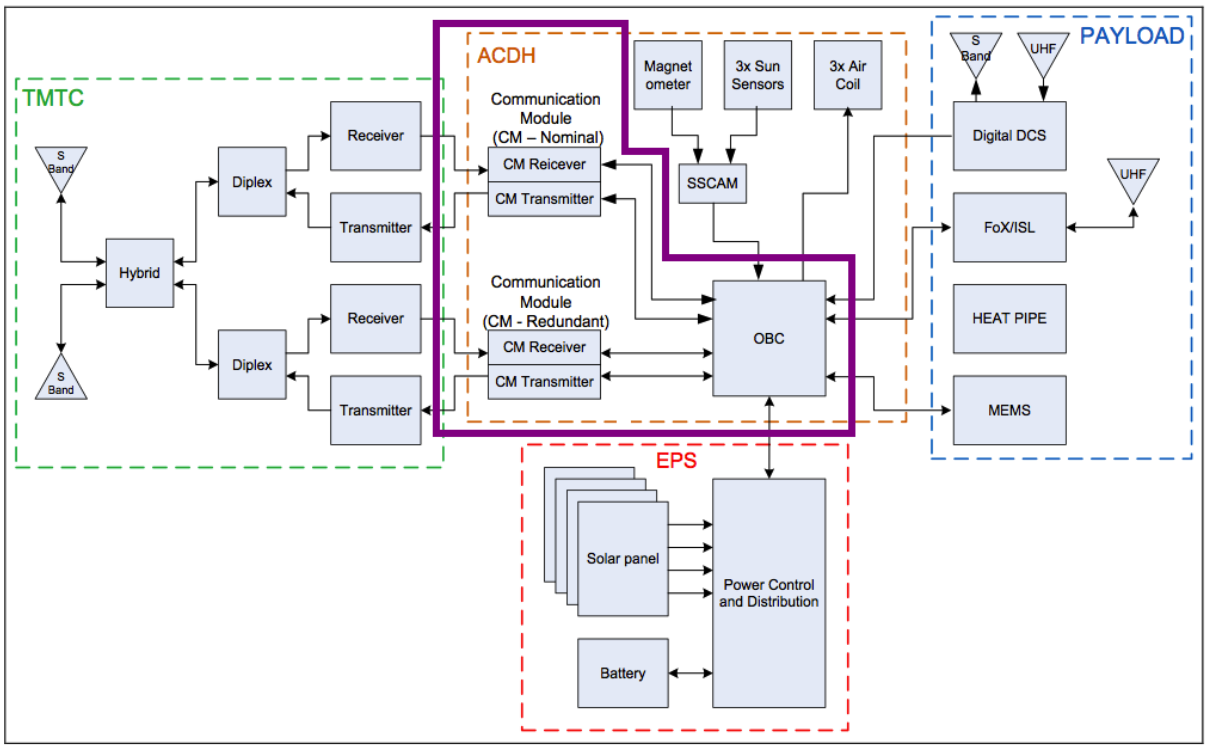

Figure 2.2: Diagram of the ITASAT satellite system

2.4 Attitude Control and Data Handling

The ITASAT project consists of several subsystems, including *Attitude Control and Data Handling* subsystem (ACDH).

The *Attitude Control and Data Handling* has two main groups of functions: satellite control and internal communication *Attitude and Orbit Control Subsystem* (AOCS), and data processing board *On-Board Data Handling* (OBDH).

There are two types of data to be handled by the satellite on-board computer: they are the *housekeeping data* and *scientific data*. Housekeeping data is information about the operation of the satellite; for example: position, temperature, electric current, voltage electrical, connected subsystems state, among others. This information is used by the ground segment to verify the functioning of the subsystems. The scientific data is the information collected by the instruments on board for studies propose; for example, images, data on the effects of radiation. The largest part of the satellite data is scientific.

Those several modules are controlled by the *On-Board Computer* subsystem, Figure 2.3, known as OBC. A 32bit processor resistant to radiation and an operating system that supports the functions performed by the satellite are used. The operating system adopted for the ITASAT is RTEMS ("Real Time Executive Multiprocessing System").

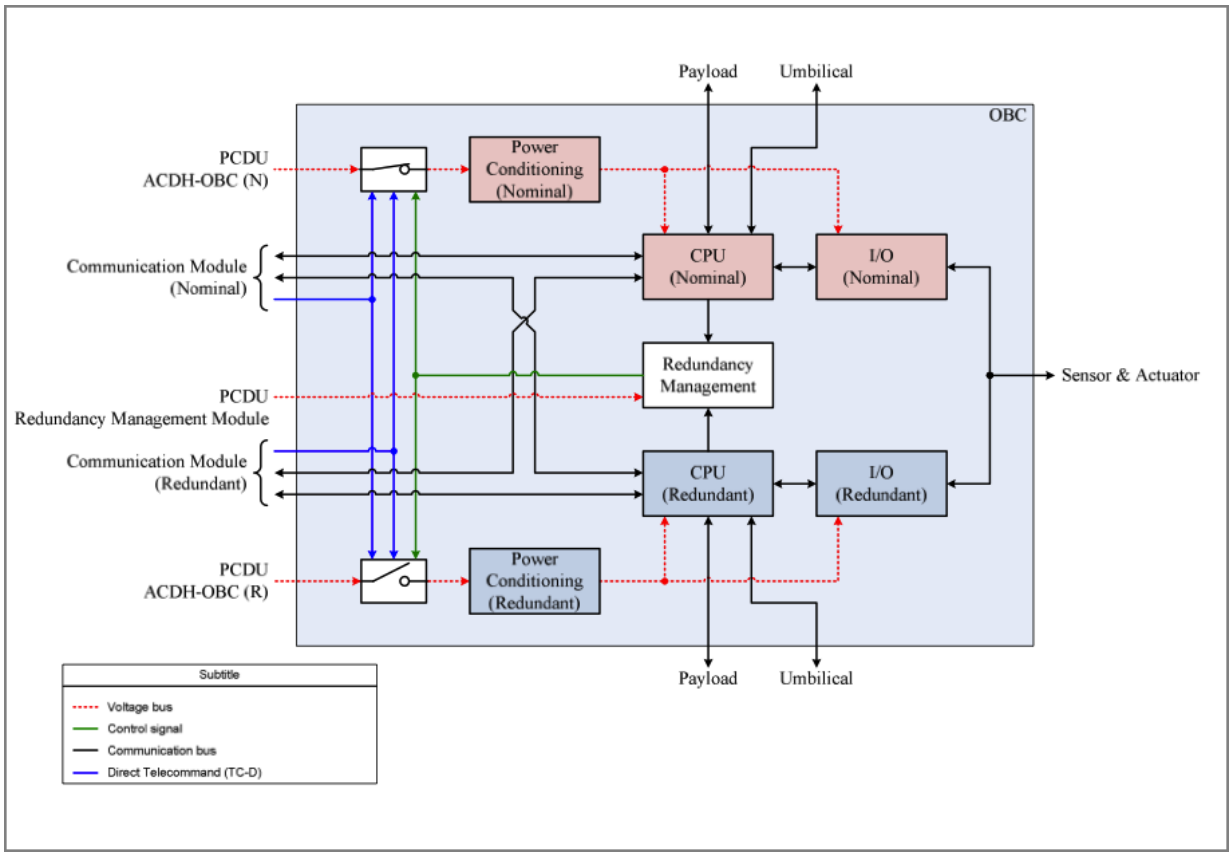

Figure 2.3: OBC Hardware Diagram

2.5 Conclusion

Brazil has a good geographic position for launching satellites and is investing in the space industry and space projects. One of this projects is the ITASAT, that was initially designed to work as a weather information center and for carrying extra payloads.

This Chapter gives an introduction to satellites technology, and the overall ITASAT project and its architecture.

The Brazilian Space Agency coordinates the overall project, while ITA and INPE ensure project execution, provides technical advise, laboratory infrastructure and financial management. This group is responsible for the complex organization of the overall mission (earth, space and communication).

The satellite architecture is divided into four main sub-systems: attitude control and data handling, telemetry and telecommand, electric power and Payloads.

OBC is an important module that combines and manage all of the other modules inside ACDH.

Chapter 3

V & V in Critical Systems

3.1 Introduction

The first vital step in a high-quality software development processes are requirements engineering [Woodcock et al., 2009]. A perfect software product is built when every step is carefully verified and validated against its specifications and, only then, we advance and transport to the next level. This way, we can be sure that the piece that is going to be used is "clean" and does not propagate error or failures. As the product development goes through different stages, an analysis is done to ensure that all required specifications are met. Software verification and validation is such model, which helps the system designers and test engineers to confirm that a right product is built in the right way throughout the development process and to improve the quality of the software product.

Verification can mean the act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether items, processes, services or documents are in accordance with the specified requirements [ANSI et al., 1995]: it is the process of making it sure that the software product is developed in the right way. The software should confirm to its predefined specifications.

Validation is confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled. Software validation is essentially a design verification function [CDRH, 2002]. Validation is, therefore, linked directly to verification [Jones and Mortensen, 1995]. Validation is the process of finding out if *the product being built is right?*. That is, whatever software product is being developed, it should do what the user expects it to do.

Validation activities include: technical reviews, walkthroughs and software inspections, checking that if software requirements are traceable to user requirements. Verification activities may include formal proofs to demonstrate logically that software is correct, however, proof techniques are often difficult to justify in non-critical or low-quality software because of the additional effort required above the necessary verification techniques of reviewing, tracing and testing.

The Verification and Validation processes work side-by-side, but visibly the validation process starts after the verification process ends. Each verification activity (such as requirement specification verification, functional design verification, etc.) has its corresponding validation activity (such as functional validation/testing, code validation/testing, system/integration validation, etc.)[Jones and Mortensen, 1995].

Two documents of airspace agencies are presented here. They proved to be very useful as formal methods guides for planning this work. The first is from the European Space Agency (ESA), and describes how to design a well structured *Verification and Validation Plan* in a critical-software project; the second presents a guidebook from National Aeronautics and Space Administration (NASA) and speaks of the importance of formal methods in software engineering and its inclusion in academic centres and specialised teams.

Formal methods have the advantage of focusing more directly on the topic, avoiding the distractions implied by implementation factors. Mechanised tools can be used to support Formal Methods in the specification and analysis phases, Table 3.1.

FM Phase	***Tool***	***Tool Function***
Specification	Parser	Checks syntactic consistence
Specification	Unparser	Translates internal representation into display and outputs formatted text
Specification	Type-checker	Checks semantic consistency
Analysis	Animator, simulator	Exhibits behaviour of system modeled by syntactic and semantically correct specification
Analysis	Proof checker, model checker	Performs proof over syntactical and semantics correct specification

Table 3.1: Mechanical Support for Specification and Analysis Phases of FM [Kelly et al., 1998]

The application of formal methods is normally divided in three phases:

Stages of the development life cycle A high level verification is made in the project requirements in the project design;

System components The verification is made through an external checking of the modules, black box testing (checking the system output, without knowing its code implementation). i.e. Evaluate how the system reacts with a given input;

System functionality This is the more low level verification, in which the internal functionalities of the module are verified.

The application of formal methods, in the first category is the most useful, because its use occurs in the early life cycle stages, given that errors cost more to correct as

they proceed undetected through the development process. It gives a first impression of the project and specification. The second category works directly with the system components, this one should be focussed only in the main and critical components due to the complexity of applying verification to a high percentage of the project. The third, ensures that the system component meets its functional specification, moreover, in some cases it is more important to ensure that a component does not exhibit certain negative properties or failures, rather than to prove that it has certain positive properties, including full functionality [Kelly et al., 1998]

3.2 ESA Document

The Guide to Software Verification and Validation [Jones and Mortensen, 1995] is one of the documents produced in a series of guides to software engineering produced in the Board for Software Standardisation and Control (BSSC), of the European Space Agency (ESA). It can work as a support guide in what a Software Verification and Validation Plan should contain. This guide is especially targeted to project managers but also useful to software engineers and software quality assurance staff.

When talking about creating a piece of software, the focus cannot be only creating a program. Developing quality software requires a very careful planning and dedication in every phase of the program life cycle. Verification and validation process is becoming an important and critical solution to achieving the desired objective. Software should be verified during every phase of its life cycle and validated before being transferred [Jones and Mortensen, 1995]. When we speak of critical software, where the process is even more important, this type of software requires greater care.

Figure 1.1 (see page 2) shows a diagram of the work flow of a project's life-cycle according to the V model. Software development starts in the top left-hand corner, progresses down the left-hand 'specification' side to the bottom of the 'V' and then onwards up the right-hand 'production' side. The V-formation emphasises the need to verify each stage (left-side) with a verifying phase (right-side), each level of production is verified against its corresponding specification. The development of a product should be a sequential set of tasks to produce a high quality piece of software. For that, a combination between development and testing is made, trying to validate each stage in order to advance to the next phase. In particular the:

Software Required Definition must be verified with respect to the **User Required Definition**

Architectural Design must be verified with respect to the **Software Required Definition**

Detailed Design must be verified with respect to the **Architectural Design**

With these verification activities, the relation between the requirements, specifications and real software produced is shown. Demonstration that a product meets its specification

should be a mechanical activity driven by the specification.

Verification of Detailed Design is efficient for demonstrating conformance to functional requirements (e.g. to validate that the system has a function it is only necessary to exercise the function). In contrast, demonstration that a product contains no defects that prevent it from meeting its specification requires expert knowledge of what the system must do, and the technology the system uses. This expertise is needed if the non-functional requirements (e.g. those for reliability) are to be met. Skill and ingenuity are needed to show up defects. The product will show more confidence to the users, if rigorous testing and verification have been made before release.

3.3 NASA Document

The NASA Guidebook from Kelly et al. [1998]is an important document that is going to support the work in terms of formal methods. It describes tools and strategies to follow to be successful on using formal methods on this same thematic of critical software.

It should be noted that there are many issues to be taken into account when considering incorporating formal methods in software development. There are many different levels of formality or rigour that can be adopted and a variety of formal methods can be used to address many different subtasks in a software development project. We have to be aware of the costs benefits of using this type of mathematics methods, it is also a matter of administrative and management factors that should be take into account. A wide variety of tools and features are available and have to be taken into consideration due to costs associated with its use. Two guidebooks are being presented: one, which addresses of management and technical support using formal methods and the other, addressing the application of the formal methods in specification and verification of software systems. It also discuss techniques,requirements and strategies for verifying high-level designs.

3.3.1 NASA Guidebook I

This Guidebook from Kelly et al. [1998] considers technical and administrative matters for the use of formal methods included in the project. It aims to include the use of formal methods in the development of critical systems, creating high-quality systems at an economically practicable level. More particularly, this volume of the guidebook is:

- written for people that are considering the use of formal methods on their project and its implications, such as project owners and managers;
- describe the advantages and the importance of using formal specifications;
- a good guideline for people that intend to plan and implement formal methods in a project. It also has tools descriptions, case studies and modifications to the levels of rigor for applying formal methods.

3.3.2 NASA Guidebook II

This Guidebook Kelly [1997] presents technical issues involved in applying formal methods to specify and analytically verify software systems. Practical techniques and strategies for verifying requirements, and high-level designs for software intensive systems are discussed. The focus is on illustrating the growing practicality of applying formal methods for enhancing the quality of aerospace and avionics applications. This work is also a product of the NASA Software Program and was cooperatively developed by a three-center NASA team formed by the Jet Propulsion Laboratory, Johnson Space Centre, and Langley Research Centre. More particularly, volume II of the guidebook is designed to:

- help the transition of formal methods from experimental use into practical application for critical software requirements and design within NASA;
- provide guidance for the novice practitioner;
- discuss technical issues involved in applying these techniques to aerospace and avionics software systems.

The Volume II also includes an illustrated example on a NASA application: the Simplified Aid, for EVA Rescue (SAFER). This example can be very useful in applying formal methods in the test plan software subsystem OBDH (*On-Board Data Handling*) of ITASAT. SAFER is a system for free-floating astronaut propulsion that is intended for use on Space Shuttle missions, as well as during Space Station construction and operation. It is a small, self-contained, backpack propulsion system enabling free-flying mobility for a crew-member engaged in extravehicular activity (EVA) that has evolved as a streamlined version of NASA's earlier Manned Manoeuvring Unit (MMU).

3.3.3 Levels of Formalisation

The levels of formalisation, as seen in Table 3.2, are defined in order to increase effort. The distinction between them, reflects the extent to which each technique is both mathematically well defined and supported by mechanized tools, yielding systematic analyses and reproducible results [Kelly, 1997]. The first uses notations and concepts derived from logic and discrete mathematics, to improve the requirements. The second uses a formalise specification language supported by tools such as syntax checkers and system simulators. This level of formality uses models and abstract data types. The third is a rigorous semantic formal specification language and correspondingly formal proofs methods that supports mechanization in this level and high specialization using theorem provers, proof checking ,and model checking and model finder.

Levels of Formalization	***Scope of the FM Use***
1. Mathematical concepts and notations, information analysis (and if), no mechanization	Life cycle phases: all/selected
2. Formalized specification languages, some mechanized support	System components: all/selected
3. Formal specification languages, comprehensive environment, including automated proof checker/theorem prover	System functionality: full/selected

Table 3.2: Levels of Formalisation and Scope of Formal Methods Used [Kelly et al., 1998]

3.4 Formal Approach

This Section presents different checking levels and different methods for applying formal methods. Several tools are considered for the different approaches. Finally, a brief introduction is made about the chosen approach and language.

3.4.1 Verification Methodology

After the system is properly formally specified, this specification can be used as a guide for developing a concrete and more reliable piece of software. It is possible to generate a model in order to find inconsistencies during the design process (i.e., realized typically in software, but also potentially in hardware). This formal specification can be made in two different ways:

operational semantics a concrete system result can be compared (which itself should be executable or able to be simulated) with the behaviour of the specification. This methodology is common called as "black-box", the verification is made by simulating the input and checking the output of the modules, without knowing the inside mechanism and how it is done;

axiomatic semantics the preconditions and postconditions of the specification may become assertions in the executable code. This is a more intrinsic verification. The verification is made on the specific functionality and make sure that if the preconditions are ensure the output is valid and respect all the requirements restrictions.

Software specification can be supported using formal methods. It is possible to write specific statements to build a guide model and to make it construct the product. A specification is a technical contract between programmer and client to provide them both with knowledge of the use cases, each describing a subcomponent of the system. Complex software systems require careful organization in the architectural structure of their components: a model of the system that suppresses implementation detail, allowing the architect

to concentrate on the analyses and decisions that are most crucial to structuring the system to satisfy its requirements [Allen and Garlan 1992; van Lamsweerde 2003].

Verification and Validation methodology is useful to make sure that the required specifications are being followed and fulfilled. This helps detect and correct mistakes and failures in an earlier state of the product. An error corrected in the beginning is much easy and low-cost to correct than in future phases.

Formal methods rigour and correctness helps to analyse and verify these models. A wide automated toolbox is available and can be used for checking completeness, traceability, verifiability, and reusability, and for supporting requirements evolution, diverse viewpoints, and inconsistency management. In this way, we make sure that what we tested is correct because we have a logical-mathematical base prove supporting it.

At implementation level, formal methods are used in code verification. It is possible to document the code with mathematical assertions and annotations, which are relations that hold between the program variables and the initial input values, in order to control the checking of the theorem or why the theorem fails to hold. Some tools can automatically generate code from a formal model.

3.4.2 Formal Approach

In order to verify if the system is working like it is suppose to, we can use the previous specification to check the system. This can be made by **Human-directed proof** or **Automated proof**. The first approach tries to prove the correctness of a program using mathematical proofs, handwritten (or typeset) using natural language, using a level of informality common to such proofs. Second approach produces proofs of correctness of such systems by automated means. The automated techniques fall into two general categories: Automated Theorem Proving and Model Checkers, that are specified using a Specification Language.

Specification Languages used to formally describe a problem using mathematical concepts. Some examples are: a formal specification notation Z; a declarative language for describing rules that apply to Unified Modeling Language (UML) models, Object Constraint Language (OCL); a language used to object-oriented and concurrent systems modeling, Vienna Development Method-Specification Language (VDM-SL); and Alloy, an expressive logic based on the notion of relations formal language.

Automated Theorem Proving in which a tool attempts to prove the properties defined in the specification. They generate formulas in a mathematical language, encouraging more reliance by the user on the system's automatic aspects.
(e.g. ACL2 [Kaufmann and Moore, 1997])

Model Checking given a model of a system, the tool checks automatically whether this model checks against a specification, see (BLAST [Kolb et al., 2010], SLAM and is Specification Language SLAM-SL [Ball et al., 2011] and Alloy Analyzer and its specification language Alloy).

e.g. Alloy contrary to other specification languages, the first approach to Alloy is "smooth". Alloy is a very simple but expressive logic based on the notion of relations. It can be used to create a first sketch of a model in a easier way. When the initial model is ready, it can be improved incrementally. Alloy was influenced by modelling languages (such as the object models of OMT and UML). Alloy's tool, the Alloy Analyzer, is a solver that takes the constraints of a model and finds structures that satisfy them. It can be used both to explore the model by generating sample structures, or to check properties of the model by generating counterexamples. Structures are displayed graphically, and their appearance can be customized for the domain at hand. The Alloy Analyser works by reduction to a boolean satisfiability problem (SAT), where it establishes if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. Several languages, such as Z, B, Event-B, UML and VDM-sl, are trying or are already translated to Alloy to make use of the its Model "Finder", the Alloy Analyzer [Mikhailov and Butler, 2002, Malik et al., 2010, Matos and Marques-Silva, 2008, Anastasakis et al., 2010, Lausdahl, 2013].

3.5 Conclusion

The ESA document is a very important standard reports of verification and validation that can be usefully to create a verification and validation plan and to use and follow some guidelines for introducing a formal methods approach to support the development of a critical software project.

Both Nasa guidebooks are very important because they can be used as supplementary survey reading material and guideline for formal methods experts, government, industry, and academia. However, the second guidebook, shows a more practical component approach to an important case study (SAFER manoeuvring device) of a formal methods application, which has been specified and analyse used the PVS specification language and iterative proof checker, used already in another NASA projects.

Nowadays, a extensive collection of tools allows us to automatically exploit many of the mathematics advantages in a formal model or formal specification. Despite their practical use in software development still being quite expensive, they can be seen as a good investment. Discovering system failures and error early on system, lead to a great reduction in future corrections. Alloy is an expressive logic based formal language that can be use to create models working side-by-by with Alloy Analyzer in order to verify and find the models. Having analyzed several options, Alloy and Alloy Analyzer were chosen for their differentiated lightweight properties in creating and verifying a formal model, and the its importance and utility in nowadays projects and research.

Chapter 4

Alloy

4.1 Introduction

Abstraction is the key to the creation of a good model for a real problem. It is important to model a system in an early stage to get a more general and concise idea of what the project really is and is going to be. Alloy was chosen for this project due to its ability to define and generate a initial model which becomes more robust and complex, evolving as the project grows. It generates a model that can be used to simulate structures and behaviours with a small part of the project description, thus having an advantage in comparison with other similar tools that need a full description, requirements or test cases. Alloy uses relational logic, defining atoms, functions, predicates and facts, those are the Alloy units. Alloy Analyzer can then use the model to check the validity of logical expressions, in an interactive way, by running and checking commands. Alloy was used because of its combination of singular characteristics:

Models and Micromodels With just a few lines it is possible to create and visualize a model and have a better perspective of the system that is being checked. Compared with other formal languages it is easy to learn and easy to use and, at the same time, very quick finding faults and counterexamples that invalidate some assertions.

Analysable Models Alloy was created and joined with a formal model checker, the Alloy Analyzer. The Alloy Analyzer is an automatic tool which allows a better visualization of simulation models without any kind of external help in terms of inputs or tests cases. It can also check properties; if a property does not hold, a counterexample is created and we can have a very helpful visual analysis.

Declarative Models A system state is defined by using a declarative approach that defines not only the state of the system, but also the actions and the transition to the next state, using predicates, invariants, properties and constraints.

Structural Models It is possible to analyse the complexity of a software system in terms of sequence of events complexity and structural complexity. There are some tools

capable of checking and analysing the complexity of those systems. Most of them analyse sequences of events, while Alloy is part of a selected group of tools that can also analyse structures complexity.

The use of micromodels is very important since the user can choose only the critical and risky areas of the full project, the core module, the most expensive modules to maintain, etc. This attention to the more important areas in a system results in a more reliable control on complex, hard-to-check projects. Alloy is a powerful formal method tool that combines first-order logic and relational calculus using a language with a syntax for structured specifications; moreover, it is capable of performing analysis with bounded, exhaustive search for counterexamples using SAT [Sheeran et al., 2000], the satisfiability problem solver.

Alloy = Logic + Language + Analysis

4.2 Atoms and Relations

Alloy uses atoms and relations to create the models and to describe the specifications. An atom is the smallest unit in the model that does not change over time and it also does not contain any extra information. Therefore, an atom is the singular, primitive entity, and it is indivisible, immutable and uninterpreted. To build a model, the atoms are *linked* by relations, which are strongly typed. Relations, sets, scalars are all the same in the Alloy language.

A relation is a set of tuples that contains atoms or sets of atoms in order to map and create more complex typed structures. In the case of a tuple with two sets, a so-called binary relation, the left part can be seen as *domain* and the right side as *range*.

The example below shows the definition (signature) of an atom `Instant` which contains a scalar (i.e. a singleton unary relation) `identifier` to store the instant id; also has two unary relations: `currentOnState` indicating the current state, and `offStates` to store the states that are not being used at the moment: in this cases, a set of atoms and not just one.

In Table 4.1 a more complete explanation is given for the relations multiplicities. The last relation is a multirelation `line`, and defines the path of the system with a set of tuples `State` → `Event` → `State`. Multirelations (relations with arity of 3 or more) are used to represent higher-order (i.e., nested) relations.

```
sig Instant{
    identifier: one Id
    currentOnState : set State,
    offStates : set State,
```

```
    line : State -> Event -> State
}
```

A relation can be described in a more abstract way without any obligation of including all the atoms connections for both sides. In order to create different relations with different multiplicities some keywords are used. In Table 4.1, the **m** is substituted with one of the four words: `set`, `one`, `some` or `lone`, each one with a different meaning and utility. If no word is used, it is assumed that it is `one`.

A **m** → **m** B

set	any number	0 .. ∞
one	exactly one	1
some	at least one	1 .. ∞
lone	at most one	0 - 1

Table 4.1: Multiplicities in a Binary Relation

Table 4.2 shows the different types of binary relations: a `function` can be represented using the word `one` in the right side of the relation (*range*), the `abstraction relation` is also extremely useful in order to model a data structure, such as an array of [A], which can be transformed into a relation in Alloy, $\mathbb{Z} \rightarrow$ A,(some $\mathbb{Z}$ have one correspondent element in A).

A **lone** → B	A **some** → B	A → **lone** B	A **some** → B
left unique injective	left total	right unique partial functional	right total surjective

A **lone** → **some** B	A → **one** B	A **some** → **lone** B
concretization	function	abstraction
A **lone** → **one** B	A **some** → **one** B	
injection	surjection	
A **one** → **one** B		
bijection		

Table 4.2: Different Types of a Binary Relation, adaptation from FM Classes

Depending on the system which is being modelled, the abstract model can be formalized using this high relations, or the more usual functions such as injectives, surjectives or one-to-one bijectives functions. In the code below, a bijective definition and a function similar to a map to extract the image of A can be seen.

```
sig Pair{ mapList : State one -> one State}

fact defValues{ Pair.mapList = (A -> B) + (C -> D) }

fun giveRange[a : State]: State { Pair.mapList[a] }
```

First, the signature `Pair` is defined with the `mapList` relation one-to-one. Next, two pairs are attributed to the `maplist` relation with a `fact` which means that is always a fact that this two pairs (A,B) and (C,D) are in the list. Therefore, it is possible to access `B` by *navigate* through `mapList` in `giveRange(A)` function. The result is the state `B`; note that maplist[a] = a.maplist.

4.3 Alloy Logic

Alloy tries to represent problems using an extended version of first-order logic that can be called relational first-order logic. This logic combines first-order logic with relational calculus. The addiction of relational operators is important in order to express more complex descriptions as transitive closure and reflexive transitive closure. Alloy translates first-order logic with a finite scope into propositional logic. In order to automate the solving problem process it tries to reduce and perform few transformations in first-order logic working as an input for satisfiability automated solvers.

Alloy allows the user to use several options when using logic to specify a model, making it even more tempting for different types of users. The example below shows several ways to express the "injective property":

Navigation expression style:	all n : Name, d, d' : Address | n → d in address and n → d' in address implies d = d'
Relational calculus style:	n → d in address and n → d' in address
Predicate calculus style:	no ∼address.address - iden

Table 4.3: Different Logic Styles

4.3.1 Propositional Logic

Propositional logic is isomorphic to zeroth-order logic and very similar to first-order logic without the quantifiers (∀ and ∃). It uses primitive symbols (atomic formulas, placeholders, proposition letters, or variables) and operator symbols that create relations between formulas. It is common used and an important logic that interprets the propositions, in this case, into two possible values: `True` or `False`. SAT tries to satisfy a formula giving a boolean formula of the primitive symbols in use to evaluate to True. With that boolean formula it can now generate a model in which the variables are generated to validate the model.
An example of a truth table used in propositional logic:

```
Conversation with 3 friends:
a) I just go to the party if Einstein and Ergos go.
b) I just go to the party if Einstein or Ergos go.
c) Because i like both, i just go out if they (Einstein and Ergos)
have/take the same decision.

p = Einstein goes to the party
q = Ergos goes to the party
```

---

	p	q	a) p ∧ q	b) p ∨ q	c) p ↔ q	
⇒	T	T	T	T	T	⇐
	T	F	F	T	F	
	F	T	F	T	F	
	F	F	F	F	T	

So the best solution is when both `Einstein` and `Ergos` decide to go out and the three friends have a nice night all together.

4.3.2 First-Order Logic

First-order logic, also known as predicate logic, is a formal system that uses constants, variables, functions and predicates to describe a system. The domain of this variables could be very large, and therefore not so easy to reduce to a satisfiability problem (SAT). An introduction of how first-order logic works:

Definition of Statements:
S1 = "Just one message can turn off the system"

Definition of Predicates:
Message(x) $\equiv$ x is a message.
TurnOff(x) $\equiv$ x turns the system off.

Logic:
$\exists$x [Message(x) $\wedge$ TurnOff(x) $\wedge$ $\forall$y [(Message(y) $\wedge$ $\neg$(x=y)) $\implies$ $\neg$ TurnOff(y)]]

"For a given message X, that turns off the system and for all messages Y that are not X, message y cannot turn off the system"
This property describes the previous statement S1.

In order to reduce first-order logic to a problem that can be decidable by a SAT solver, it is necessary to apply some inferences rules. Among them we cite as an example the Universal Elimination or Existential Introduction: adapting the `forall X` to a more restrict domain of `(x1, x2,..,xn)`. Universal Elimination reduces the universal domain to a more restrict and small domain, eliminating the infinite scope (`for all`, $\forall$); Existential Introduction gives a "name" to a variable that exists in the formula to make the proposition formula always true. The quantifiers elimination is done in order to calculate the solution with a more practical scope.

4.4 Alloy Language

The Alloy formal specification language can be considered more light and easier to use than other languages for the same purpose such as Z [Jackson, 1995] or Vienna Development Method-Specification Language (VDM-SL) [Bera, 1988]. The most basic element in Alloy is the atom that can be defined by `sig atom{}`. An atom can have several relations inside it.

These Alloy structures can be very similar to Java in terms of objects inheritance. Different kinds of states often have a certain amount of characteristics in common with each other. In the example below, the commands: `BinaryCommand`, `SynchronousCommand`, and `MessageCommand`, all share the characteristics of an abstract `Command` (current state, binary attribute or string id). Yet each also defines additional features that make them different: Binary command can send a extra bit to the system, the Synchronous command have two objects to be synchronise and Message command have a string that contains the message.

```
abstract sig Command{
    currentState: one State,
    active: one Bool,
    id: one String
}
    abstract sig routineCommand extends Command{}
    sig BinaryCommand, SynchronousCommand extends Command{}
    sig MessageCommand extends Command{}
    one sig UniqueCommand extends Command{}
```

This can be used for choosing a direction in a system path, i.e. `yes` or `no`, `messageType1` or `messageType2`. An `abstract` state is a state that is declared abstract, it may or may not include abstract relations. Abstract states cannot be instantiated, but they can be subclassed. The keyword `one` is used to defined that the atom is unique in the model. i.e. `# UniqueCommand` $>$ `1`, (the cardinality of a unique command is greater than 1) is always false.

4.4.1 Operators and Quantifiers

In Alloy everything is a relation, relations can be joined with operators in order to create a more complex relation. There are three types of operators: `set operators`, `relational operators` and `logical operators`.

Set operators include the usual operations such as `intersection` (&), `union` (+) or `difference` (-). The '=' is used to compare if two sets are the same. The keyword `in` checks if a set is part of another. For example, *"a in b"* and *"b in a"* is the same as *"a = b"*.

Relational operators include operators such as *join*(.), *transpose*($\tilde{}$), *transitive closure*($\hat{}$), *reflexive transitive closure* ($*$), and *product* ($\rightarrow$). The *join* operator works as a navigation operator, for example:

```
if:
p = (A -> B)
(p defines a relation A to B)
q = B

then:
p.q = A
(the domain A is returned, because a navigation
is made from B to A in the [A,B] set relation.)
```

The transpose ~r of a relation r is its mirror image, reversing the order of the mapping. The transitive closure ^r of a binary relation r is the smallest relation that contains r and is transitive. The reflexive, transitive closure *r is the smallest relation that contains r and is both reflexive and transitive. i.e., it is similar to ^r but includes a mapping from every atom to itself. The product of two relations is similar to the join, but the intermediate atoms are not dropped. The result is the relation containing the concatenation of every tuple in the first relation with every tuple in the second relation [Crane et al., 2003].

Logical operators includes the operators that are used in formulas. The operators are similar to the *set operators*: conjunction (&&), disjunction(||), implication(⇒), biimplication(⇔) and the negation(!). Quantifiers are also used in formulas that contain free variables. Alloy has five types of quantifiers as illustrated in Table 4.4.

Quantifier	Meaning
all x : e | F	universal, F is true for every x in e
some x : e | F	existential, F is true for some x in e
no x : e | F	F is true for no x in e
sole x : e | F	F is true for at most one x in e
one x : e | F	F is true exactly one x in e

Table 4.4: Alloy Quantifiers

4.4.2 Functions, Facts, Assertions and Predicates

Functions Function definitions can be seen as formulas. The functions receive some arguments and return the set of the results that match with the written definition.

```
sig Pair{
   pair: Left -> Right
}

fun theOtherSIde[p : Pair]: set Left {
    p.pair.Right
}
```

The objective of this function is to get the `Left` side of the `Pair` relation. It receives the object `Pair`, and returns a set of `Right`, it does a navigation from `Right` in the `Pair.pair` relation.

Facts are similar to signatures but are used to define a model "constant" attribute that cannot change over time and as to be always satisfied. They can be written in a separate paragraph or be a part of a signature. In the example below the '$\#list <= 4$ fact is written under the A signature, and it works such that the cardinality of $this.list$ has to be less or equal than 4. Meanwhile `fact1`, because is written in a different paragraph, does not have the association of the signature, for this reason is necessary to add the `A` in the beginning of the expression.

The final model is the conjunction of all facts and invariants. As is shown is Figure 4.1, the Alloy Analyzer creates a model following the model description. In this example, the model must have a `state A` with connections to three states (because of fact1), plus two extra states resulting for the `run` command, explained in 4.4.3, requiring the model to have five states.

```
sig State{
}

one sig A{
    list : set State
}{
# list <= 4
}

fact fact1{ # A.list = 3 }
```

```
run{} for exactly 5 State
```

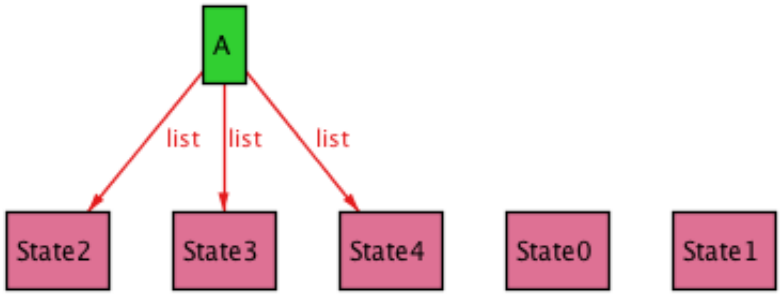

Figure 4.1: Conjunction of facts ($list <= 4 \wedge list < 3$)

Assertions are very similar to *facts*, but instead of being used in specification, they are used during the model analysis. They have to be written in a separated paragraph in the specification phase and them in the checking phase it is checked if the specification satisfies the assertion expression. If not, a counter-example is showed to demonstrate where the model fails. An assertion represents a model limitation (e.g. a the list should have exatly 3 elements), and if the verification fails, it means that the model can have a non-intended configuration.

Assertions are very useful to find errors in the model and in the specification, if an assertion fails it means that the model was not suppose to accept a new instance but it did, so some work has to be done to fix the incoherent model.

Predicates To express the requirements in the documentation as a model in Alloy, predicates can be defined. Using the keyword `pred` it is possible to express predicates and restrictions that we want the model to guarantee. This verification can be done associating the predicates to the check command, see 4.4.3. The Alloy Analyzer tries to find a counter-example within the specified scope. In the case that the assertion is false, it will show an instance of the counter-example model and the user can understand what happened and which were the steps which lead to this unwanted state. If the statement is valid it will present a "No counterexample found. **initialState** may be valid.".

These sentences guarantee that for every possibility in this model within this scope (for this specific domain) the formula holds; still, it does not say or prove anything if the scope is extended. In the case that a rule has to be declared and it is a constant in the model, it

can be written as `facts`. These statements cannot be changed during the model execution; however, careful attention should be paid when applying this facts in order not to restrict the models by rules and not for its own structure.

In the example below, a signature of a `Counter` that has just a value is defined; this value is incremented 1 or 2 units in each iteration.

```
sig Counter{
   value : Int
}

fact transition{
   all t :Instant , t' : instant/next[t] | one i : PLUS_1or2[] | next[t,t',i]
}
```

Suppose that the user wants to verify if, at some point, the value reaches an undesirable number. The counter initiates at 0 and cannot equalize 5. The implementation should be something like: "if(3) then(+1) else if(4) (+2) else (random sum)". If somewhere in the model description the following fact occurred:

```
fact {
   no c : Counter |  (int c.value = 5)
}
```

the counter would never reach the value 5, not because of the description or the system functionality, but for a stronger restriction that limits the creation of the possible instances.

The main difference between facts and assertions is that facts are restrictions to the model and assertions are a specification that needs to be verified. The assertions can be verified using the Alloy *check* command.

4.4.3 Run and Check

There are two commands in Alloy: `run` and `check`. The command `run` is capable of generating instances according to the specifications in the Alloy language. The Alloy Analyzer is capable of generating every non-isomorphic instances within a defined scope. Using the 'Next' command is possible to get another instance from the same model.

One possible way to look for errors and flaws or even verify that certain constrains and variants are maintain during the model is to create *assertions*. The check command checks the specified model against the defined specification.

```
sig State{
    list : State
}

fact contains{
    all s: State | s in s.*list % and s not in s.list
}

assert cycle{
    all s: State | s not in s.list
}

check cycle for 3

run{} for exactly 3 State
```

It verifies if the condition defined in the assertion holds for every instances in the model, trying to find a counter-example that in a specific case satisfies de specification and yet fails to satisfy the assertion.

Figure 4.2 is the result of running the check command for the 'cycle' assert,a state cannot have a direct link to itself. The Alloy Analyzer has found a counterexample that invalidates the assertion. In case of including the same restriction as a fact, Alloy would recognized this fact as a model definition and include it as a model specification, becoming a strong restriction. Now, if the check command is ran again it will show "No counterexample found. Assertion may be valid." because the analyzer cannot find a assertion or rule violation since the model does not accept nodes with self-recursive links, the condition holds for this new model specification.

4.5 Alloy Analysis and Analyzer

First-order logic is undecidable and because Alloy uses this logic is necessary to do some adaptations. In order to make it decidable Alloy uses the first-order logic with a finite scope `k`. This small trick allows a satisfying instance in scope with no more than `k` atoms of each type. It is necessary to pay attention to the scope size when creating a model due to is exponential explosion. However, because the analysis is restricted in scope, there

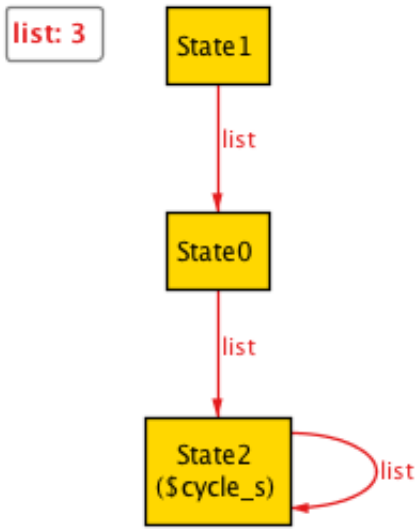

Figure 4.2: Checking assertions with Alloy `Check` command

is no guarantee that a counter-example would not be found with a larger scope. On one hand, the decidability for a finite scope is gained, but on the other, the complete analysis is lost.

4.5.1 Alloy Analysis

The analysis of Alloy depends on the translation of the first-order logic or relational logic due to its powerful addition of relational calculus in propositional logic (Boolean formula). It takes some steps before the SAT solver can find some model.

- Translate the higher logic, for a chosen scope, into a propositional logic formula;
- The mapping between relational variables and boolean variables are preserved;
- Convert the boolean formula in a conjunctive normal formula (CNF).
- CNF can now be used as input in the SAT solver.

Still the SAT problem is, in computer complexity theory, a well known non-deterministic polynomial-hard (NP-hard problem). This problem is untractable because it is a 2^N algorithm. When modelling a problem in Alloy an extra consideration should be taken in terms of space scope. The calculation suffers from what we could call the ‘‘`double when plus-one`’’ rule, that means that the scope is doubled when a new variable is added:

$\times 2 = +1$ ‘‘`double when plus-one`’’

This rule is applied because the truth table used to perform analysis is exponential. When an extra variable is added to the system, the possibilities double, Table 4.3.1 shows that if a new `r` variable is added to `p,q` the table increases its size in double. For this reason Alloy does not scale, and in the future will also be restricted to small scopes.

4.5.2 Alloy Analyzer

The Alloy Analyzer has some limitations/restrictions when a more large scope of constrains is necessary. However some studies [Jackson, 2000a] were made proving that the default scope of 3 is normally large enough to find most of the errors. Results demonstrate that Alloy is efficient at analysing specifications with small scopes, with the default scope of 3, models can be analyzed in well under a minute.

The small scope hypothesis postulates that it is possible to find practically all errors of the problem using just a small scope, but it makes sure that for this space solution the asserted property holds for every situation. It is possible to say that we loose in quantity but we win in quality. This hypothesis is useful in practice when developing specifications; however, it is a purely empirical hypothesis.

Another situation, is when trying to satisfy a problem that, for a scope `k`, there is a specification that has no satisfying instance in `k`, but does have one in scope `k + 1` [Jackson, 2000b]. In this case the model domain had to be enlarge for satisfying the problem restrictions.

4.6 Conclusion

Alloy is a formal language that can be relatively easy to start with, it is very efficient in giving the user a quick and early idea of the general project state.

Its capability and advantage in creating models in an early stage, and iterating them to built a more complex and robust project shows that Alloy can be a very useful tool and problem finder. Some of its advantages are: using set theory and relations translates first-order logic in proposition logic with a finite scope; translates specifications to formulas and solves them generating friendly visual models that can be still explored using the evaluator to check the evolution of the individual atoms, thus checking the progress of a variable during the project execution.

However, Alloy can be only applied in problems with small scopes due to is truth table method exponential algorithm restriction. In order to use Alloy verification efficiently it is necessary to focus the efforts in dividing and abstracting the problem in smaller pieces and the work in the more crucial modules in the project development.

Chapter 5

Formal Model Template

5.1 Introduction

The most important part of any project is the planning part, since, without a good design, the project can take unexpected and unwanted directions. A unaccurately prepared project may cause unexpected critical problems and, depending on the project, could mean losing big amounts of money or even human lives.

In order to reduce these problems, a big effort should be made to ensure that the design is correct. In this case, we are working with projects whose design can be reduced to a dataflow or path decision diagram (a diagram that need a valid transition to proceed) and we want to be sure that some requirements are always valid during the project execution.

In order to perform such a verification, we are using Alloy, a lightweight formal language, which is capable of modelling the project using relational logic and which supports making formal assertions in the created model. Alloy is also a very useful tool because it can create a model in the earliest part of the planning stage and provides a quick view on the directions the project is going to take.

We divided our work in three phases: "Formal Model Guidelines", "ModelMaker Automatic Building" and "Checking the Requirements":

In Phase 1 -"Formal Model Guidelines" a modelling strategy was elaborated in order to create a behavioural formal model, starting with an initial one and translating the requirements to facts in Alloy; then the model was enhanced, making it capable of working with flags, synchronization and interruptions.

Phase 2 -"ModelMaker Automatic Building" included the creation of a tool capable of creating a model description in an Alloy file, which follows the rules previously defined in Phase 1. This Alloy description can automatically and quickly generate a model that can be formally analyzed with just a few instructions and the configuration of the tool.

Phase 3 -"Checking the Requirements" applies the work made in the first and second

phases to a critical real case, the ITASAT project. The project was sub-divided in four stages. We followed an approach which starts at a higher level and more abstract stage to a lower and more concrete level. We named it "System-To-Code Approach": it starts modelling the `System Modules`, then `Software Modules`, Communication Module until a lower stage such the code itself, represented in this example as `Receiver Code`.

5.2 Phase 1: Formal Model Template

Our objective is to create a structured model to work as a design checker template to make the process of modelling more efficient and less time consuming. The abstract general idea is to give a form to the user so that he can fill it and successfully create a model in a short period of time, start reasoning and have some formal assurances about the project. The presentation of the template is divided in two parts: the first part is about creating and simulating a model, while the second part adds some features to the initial model such as synchronization, interruptions and flags.

5.2.1 Initial Model

Our focus are systems that can be represented by state transitions which respond to commands and can change the overall state. The first step in order to create a initial model is to obtain the essential information about the design. Our model is based on transitions and the way that they affect the system state. This transitions are captured in the project restrictions and requirements. With this transitions information, that in fact are state possible events from the current state, is possible to create the tuple `State->Event->State` that is the necessary transformation rule for a state evolution. These events can be seen as a simple next/straightforward event (does not have any associated restriction) or a more complex structured hierarchical transition. As shown in the following example, `Message` is defined as an abstract signature and the `reboot` message extends `Message`, that is hierarchically higher.

```
abstract sig Message{}
one sig reboot extends Message{}
```

As shown in the example below, the main signature was named as `Instant` and defined as a ordering module due to its import role over time. The "object class" `Instant` is going to be the "engine" that simulates the changes through time in the model, representing moments in time. A further explanation about this behaviour of the overall state "screenshots" can be seen in the subsection 5.2.3.

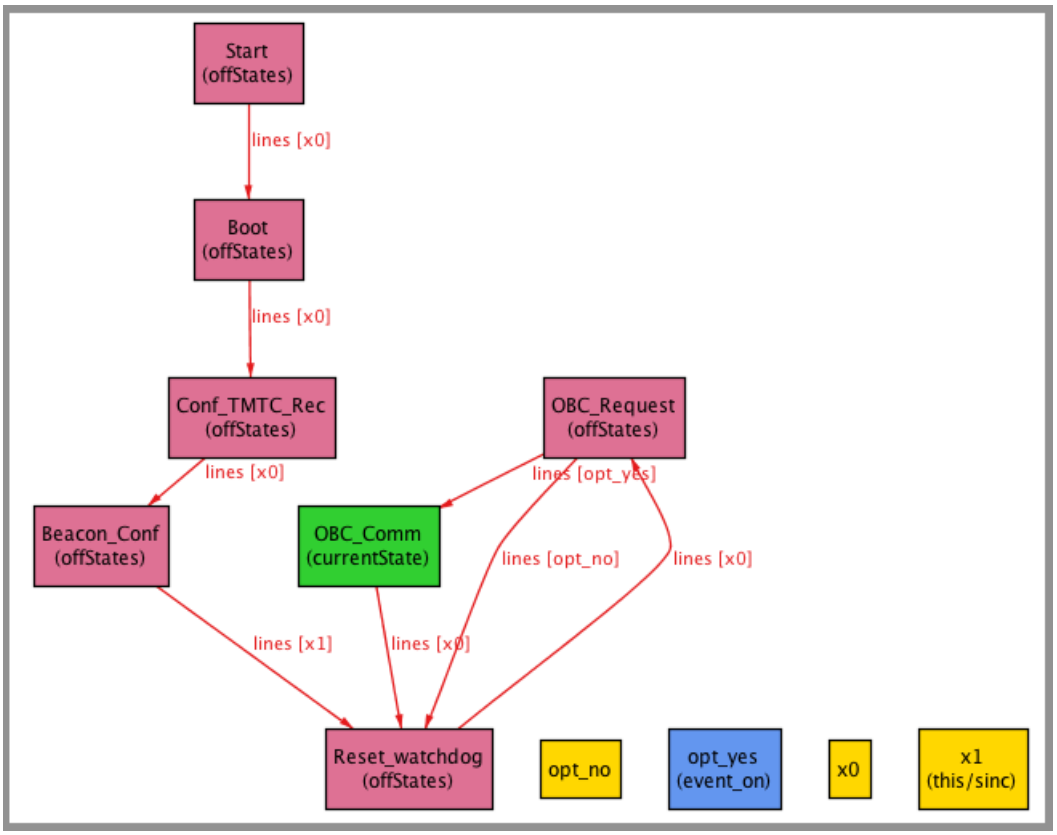

Figure 5.1: Initial model with events

Figure 5.1 presents an example of an automatically generated Alloy model which uses the guidelines defined in this work for creating an initial model. The `currentOnState` and `offStates` attributes are defined to give state information about which state is currently selected. This is an important attribute to check if, at a given instant, the systems allow or not a transition for two different valid states from the same event. An example of that is the event "sending a message from a valid transmitter", that violates the "just one destiny" rule if both transmitters are in a valid state. This two attributes also allow model color differentiation and offer an improved visual observation and sense of movement when the system state changes. The `currentEvent` is defined in order to know which events, randomly chosen from a set of possible events, have been chosen by the model in a specific stage.

```
open util/ordering[Instant] as instant

sig Instant{
   currentOnState : set State,  --current state
```

```
   offStates : set State, --off states
   currentEvent : one Event
}
```

Next, the `State` and `Event` classes are defined. All states are declare under the State signature. In order to declare a sub state, it is necessary to create a new abstract State, since every state just allow a singular atom creation. `sub_StaTe2` is an example of this situation.

Similar to `State`, every `Event` have to be declare under the abstract class `Event`: the Alloy language allows inheritance in classes, which allows the creation of different types of events; in this way, more complex restrictions can be applied in the transitions to make transition decisions more efficient. Filter is the action of filtering events from the current list of possible events. As shown in the example below, the `x2` signature can be used as a sub-class of `Access` taking advantage of the class hierarchy, that means that every event that allows `Access` will also allow `x0`, `x1` and `x2`.

```
abstract sig State{
   lines : Event -> State
}

one sig Start, Boot, (...) extends State{}
abstract sig StaTe2 extends State {}
one sig sub_StaTe2 extends StaTe2

abstract sig Event{}
abstract sig Access{}
one sig x0, x1, x2 extends Access{}
```

The transitions and the initial states are defined using `fact` statements. The description of the possible transitions is written in a list of tuples of three elements: the origin, the valid event and the destination. All of the system states are declared as `State` atoms. The main flow or system path is defined in a attribute called `lines`.

```
//define transitions
fact transitions{
```

```
    lines =
    Start -> x0 -> Boot
    + Boot -> x0 -> Conf_TMTC_rec
    +(...)
    + OBC_Comm -> x0 -> Reset_watchdog
}
```

This `lines` definition, uses the abstract `State` declaration, that specify that a state has a relation of (event,destination). Since in Alloy everything is a relation, in the end, the system path is defined in `lines` as a list of tuples (origin,event,destination).

5.2.2 Requirements and Facts

When a model verification is done, it is necessary to validate the initial State and from it all possible valid State transformations. To ensure and maintain the formal verification in the model it is necessary to define some characteristics such as: initial State validation properties, problem invariant and transition invariant (i.e. from a given valid `State` and a valid transformation, a new valid `State` is reached).

```
pred initial[t : Instant]{
    t.currentOnState = Start
    t.offStates = State - Start
    (...)
}

pred invariant[t : Instant]{
   one t.currentOnState
   t.offStates = State - t.currentOnState
   (...)
}

assert initialInstant{
    all t : Instant | initial[t] => invariant[t]
}
check initialInstant

assert transitionPreserves {
    all t, t': Instant |
      some event : possibleEvent[t] |
```

```
        invariant[t] && move[t, t', event] => invariant[t']
}
check transitionPreserves
```

The `initial` predicates, define some properties of the initial `Instant`, while `invariant` predicate, declare some `Instant` properties that should not change over time. The assertion `initialInstant` verify that if the initial state is also a valid one. By checking the `transitionPreserves` assert we can verify if, starting from a valid state and using a valid `Event`, a state that does not preserve the invariant can be reached, and so, not fulfilling all the pre-establish requirements.

Predicates can impose certain initial and final restrictions. `Pre` and `Pos` conditions are used in operations and functions for filtering the input and validate the output. These conditions work as input restrictions and output insurance. For example, if an operation is defined to add a message in a memory that does not allow replicas, the message needs a verification in order to check if the message does not exist in the memory and if the memory has space to store this new message. In the end, the memory counter should increment one unit and the memory should have the new message in its new instant.

```
pred add[mem,mem':Memory , new_msg:Message]{
   --pre conditions
   new_msg not in mem.messages => mem.pointer < mem.size - 1
   --pos conditions
   mem'.messages = mem.messages + new_msg
   mem'.pointer = mem.pointer + 1
}
```

In this way the operations are no longer total: their domain has been restricted. The scope of the domain is restricted and we modelled a more safe function. Another example is the divide function that cannot compute with 0 as the dominator. In this case we should put a pre-condition for the imputed variable $! = 0$.

5.2.3 Move your Model

To simulate the model behaviour, it is necessary to create an ordered relation between the different `Instants`. This linked atoms that simulate the behaviour were named as `Instant`. It is possible to use the Alloy *ordering* module to impose an ordering on a set of `Instant` atoms. A new `Instant` is created when an action in a previous state occurs, being

necessary to connect and order these `Instants`. This action does not necessary imply a change.

```
open util/ordering[Instant]
```

The "util/ordering" module connects the instants assigning an ordered relation between the atoms, also providing a few useful functions that can be used, including `first`, `next` and `last`. When applied with the ordered atoms, `first` and `last` give the first and the last atom respectively, and `next` gives the adjacent higher atom in the linear ordering.

Now that util/ordering has been loaded, the initial atom `Instant` can be defined in order to set the model initial state.

```
--initial state
fact init{
    let t0= Instant/first |
        let initInstant = Start|
            t0.currentOnState = initInstant &&
            t0.offStates = State-initInstant
}
```

The first `Instant` (Instant/first) is renamed to `t0`, and its `currentOnState` relation gets the initial state (`Start`). This state `Start` is the first state in the model and every possible different configuration starts from here. The `offStates` gets the other states except `Start`.

In order to move the model, two basic states are needed: the current state and the next state. First, a set of possible events from a current state for a certain `Instant` is calculated, then an event in chosen from that set. Next, the destination is calculated with the current state and the selected valid transition. At last, the new instant(X+1) is calculated from the previous `instantX` with the necessary alterations and extra information. Doing this iteratively, it is possible to move and simulate our model.

The set of possible events are generated in the function `possibleEvent`. The function receives an argument `Instant` giving the information of the current instant of the model. The possible events are taken from the list of tuples "lines[State]" that show the current state in the left argument. Then, to make the model "move" the `moveM` predicate that receives three arguments is defined, the `instant` that is the current instant, the `instant'` that is the next instant and the `event` that was previous and randomly selected form the set of `possibleEvent`: now, the next state (`currentOnState`) that is going to be chosen in the list of tuples from the current instant `t` (State→Event→State) by the pair (current state, event) can be calculated. The `turn_off` becomes the new current state. At this

point, it is just necessary to upgrade the new `Instant`, defining the new alterations and also maintaining the attributes that did not suffer any alteration. Thus, we have the following signature for our transition function:

```
fun possibleEvent[instant:Instant]: set Event{
   lines[instant.currentOnState].State
}

pred moveM [instant,instant':Instant, event:Event]{
   let turn_on = (lines)[instant.currentOnState][event] |
     let turn_off  = instant.currentOnState {
       instant'.currentOnState = instant.currentOnState - turn_off + turn_on
       instant'.offStates = instant.offStates + turn_off - turn_on
       instant'.currentEvent = event
      }
}

fact stateTransition{
    all s :State , s' :state/next[s] |
     some event : possibleEvent[s] |
        moveM[s,s',op] or moveM2[s,s',op]
}
```

Figure 5.2 shows four iterations of the model. The green color in the `States` shows the current selected state and the used transition event to reach the selected state. The yellow color shows the states that are not being used at the moment and the not selected transition events. The example shows the behaviour and evolution of the model from the state `Beacon_Conf` until `OBC_Comm`. Sub-image 2 shows that the event `X1` was chosen and the currentState is `Reset_watchdog`. In Sub-image 3, the chosen event was `X0` and it arrives to a state `OBC_Request` that with an event `event_yes` evolves and reaches the `OBC_Comm` state again.

5.2.4 Synchronization

To defined a synchronous event we need to define a **sinc** relation to store the synchronized actions list. This special events need to be simultaneously executed in both systems. This feature has great importance when modelling systems that have sub-systems that communicate with each other. For example, a main processing state wants to send a message and the communication module has to be in a "sending" state. The example below

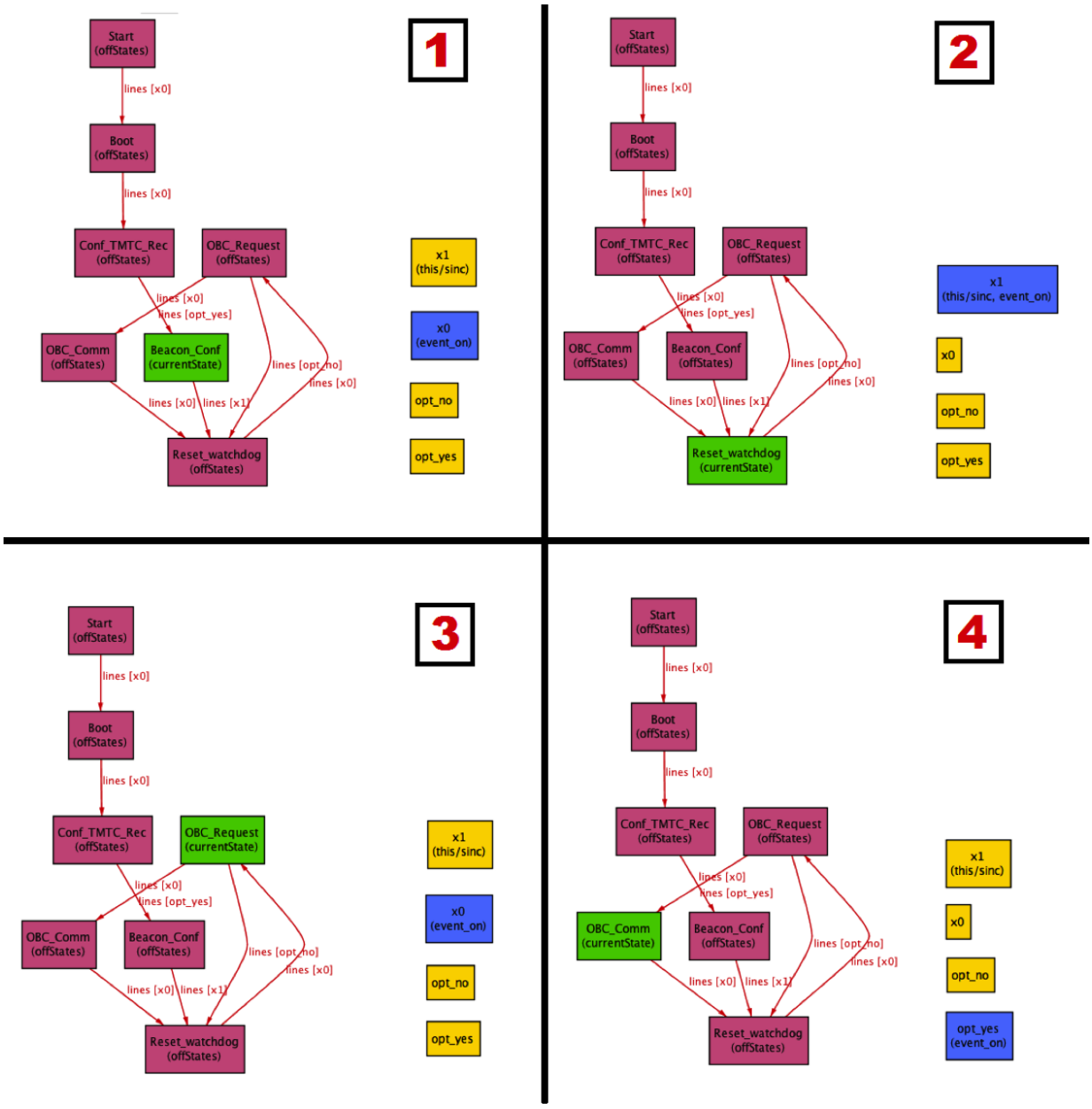

Figure 5.2: An example of dynamic modelling simulation in Alloy model.

shows the definition of a new synchronized event `x1`:

```
sig Instant{
    currentOnState : set State,
    offStates : set State,
    currentEvent : one Event
    sinc : set Event --NEW
}

fact {
//define sincronized actions
Instant.sinc = x1      --NEW
}
```

Now, in the model, it is necessary to add this function in the `possibleEvent` function, to restrict the available events set. Figure 5.3 shows a synchronized example. Model A and model B where `x1` action is a synchronized event, if the system is not simultaneously at the same state the flow will stop until this condition is reached. If a possible state is reached where the model A is in `a2` and the `x1` is the only possible event, that system has to wait until the system B reach `b1`. In this scenario, both systems evolve and change the current state to `a3` and `b2` respectively. In the example below, `possibleEvent` is reused and added the possible synchronized events for the current instant. Function `possibleEventSinc` is defined to capture only the events that are defined as synchronous and can make a transition from the current state.

```
fun possibleEvent[s:Instant]: set Event{
   let event = s.sinc  |
      (lines[s.currentOnState].State - s.sinc) + possibleEventSinc[s,event]
}


fun possibleEventSinc[t:Instant,event : Event]: set Event{
--PRE @event must be "event = t.sinc"
   let b_on = (lines).State.event |
      let result = lines[b_on].State |
         b_on in t.currentOnState => result else none
}
```

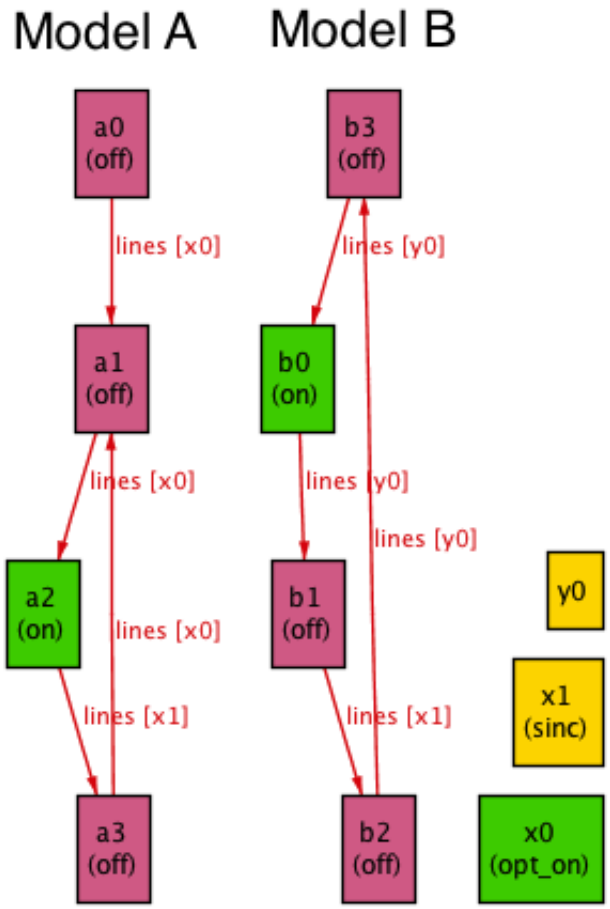

Figure 5.3: Action Synchronization in model A and model B

5.2.5 System Exceptions

The exception feature is implemented to enable the modelling of interruptible systems. The normal behaviour of an exception is to:

- Interrupt the main process.
- Execute the exception until it is finished.
- Return to the same instance in the interrupted process.

Figure 5.4 shows a simple exception model (an exception model that cannot be interrupted). In this example, the main model (model 1) receives an interruption flag (bool = True), then, stop the main flow to answer the system with higher priority. Model 2 reaches the state `Z` and only now the main model can continue with its usual flow. This

is important for studying behaviours and to make system verifications. For example, in a communication process, an interruption can change the state in the middle of a message with tragic consequences. With this model, we can control and predict how the system is going to react to a possible interruption and, also, verify its robustness in the event of a unexpected behaviour.

The Alloy Exception Model has the following structure: The signature `Instant` has now three extra variables: `on_Exc`, `off_Exc` and `bool_Exc`. The first two are added in the other Exception Model specification: for a similar utility than `currentOnState` and `offStates` state sets are defined in the main model, to store the current state, and also to store the states that are not being used.

```
sig Instant{
    (...)
    onExc : set State,
    offExc : set State,
    boolExc: one Bool,
}
```

The `bool_Exc` is the signal that is randomly generated in every iteration of the system. This boolean implies an interruption in the main model to answer an exception. The predicate that moves the model changes in order to be capable of evaluating the exception variable before calculating the next state. Figures 5.4 show that the main model (Model 1) is interrupted in the middle of a routine and priority is given to answer the model exception (Model 2). The processes between the main model and exception model are very similar; it is only necessary to check carefully that the exception model runs until its end and remember that just the main model is capable of generating a random boolean.

5.2.6 Booleans, Integers and Flags

Boolean

At some point in the design is necessary to make a binary decision in the model, otherwise is just a linear transition flow that does not to evaluate any decision value. Alloy has no native boolean type, because it increases the complexity of the language and the user can make a bad interpretation of a boolean type. If a boolean type is imported, it would appear as in the following signature, where an attempt to express the following requirement is made: “A Satellite cannot be on earth and in space at the same time”:

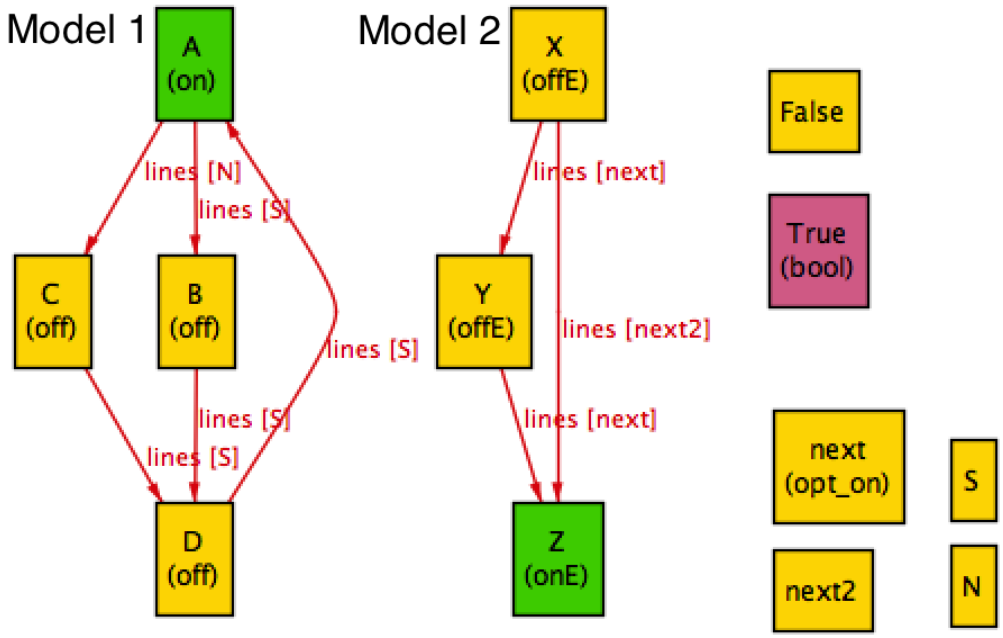

Figure 5.4: Main model and Exception model

```
sig Satellite{space, earth : boolean}

fact{all s: Satellite | s.space => not s.earth }
```

Because of the set theory, the relations `space` and `earth` are in general a set of boolean. That could lead to trouble if the set is empty or its cardinality is higher than one. So, we used another approach to represent the boolean types [Jackson, 2012].

```
abstract sig Boolean {}
one sig True, False extends Boolean{}

sig Satellite {space, earth : one Boolean }

fact no_paradox {
```

```
    all s:Satellite | s.space != s.earth
}
```

Now, with this new signature it is possible to describe the previous model, and since the relations have, by obligation, one boolean constrain (True or False), it is possible to make a fact defining the Satellite requirement.

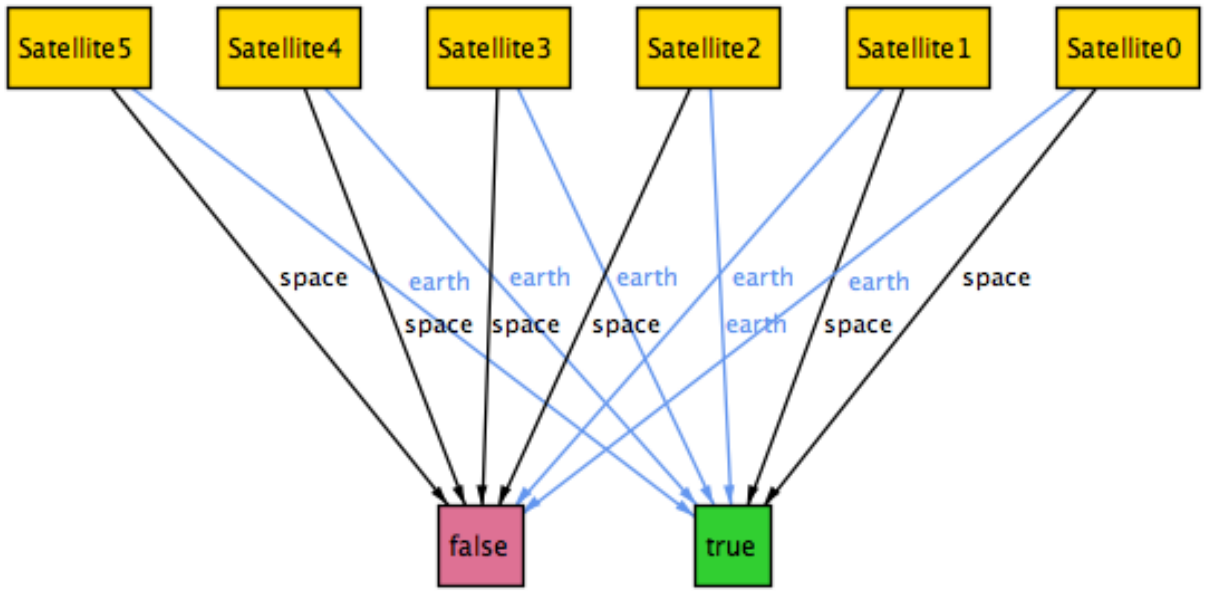

Figure 5.5: "A Satellite cannot be on earth and in space at the same time", Bool example

Figure 5.5 shows a boolean example in which a `Satellite` cannot be on `earth` and on `space` at the same time. This is an example of how an adaptation is used to work with boolean constrains. **Flags** can also be defined in a similar way and be used to mark or designate some structures to decide the model flow.

Integers arithmetic

Integers are not atoms, but combined with each *integer value* there is an *integer atom* that holds that integer value, allowing the integer effectively to be inclued in a relation [Jackson, 2012]. Integers can be combined with addition, subtraction and comparison. Still, if more functions with Integers are wanted, `open util/integer` can be used in order to have more than ten new Integers functionalities such as *grater than*, *nexts* , *max*. To

increase the scope of possible atoms number used we can define an extra scope specification `Int` in the run command. In this example, the scope 5 Int assigns to the signature Int the range of integers from -16 to $+15$.

```
run {} for 5 Instant, 5 int
```

However, the use of integers should be avoided, since it needs to represent all the atoms until it reaches the sum of the desirable number, and the model can become too heavy and not treatable due to the full representation. The use of Integers makes the Alloy model even less scalable. In this example, we added the feature Counter, that increases its value with a given function `PLUS2`.

```
pred next[t,t' : Instant, i : Int]{
    int (t'.flag).counter = int (t.flag).counter + i
}

fact transition{
   all t :Instant , t' : instant/next[t] | one i : PLUS2[] | next[t,t',i]
}
```

Now, defining a model and using counters and booleans, a better management of the system derives; when the counter value reaches some value, it makes a different action from the one previously defined. Figure 5.6, the result of the example above, shows the increasing value of the `Counter` throughout instants. It starts with the value 1 and increments 2 in every transition through instants.

5.3 Phase 2: ModelMaker Automatic Building

ModelMaker (MM) is a tool that helps develop an initial formal model in a quick way. Our goal is to improve and introduce more quality in the modelling process, reducing the human error when creating a formal model of the system. This tool was created to speed up the validation process and to allow for a first model to be developed in the shortest period of time. The output is a formal model created in Alloy that can be used to check requirements, according to the guidelines defined in Phase One.

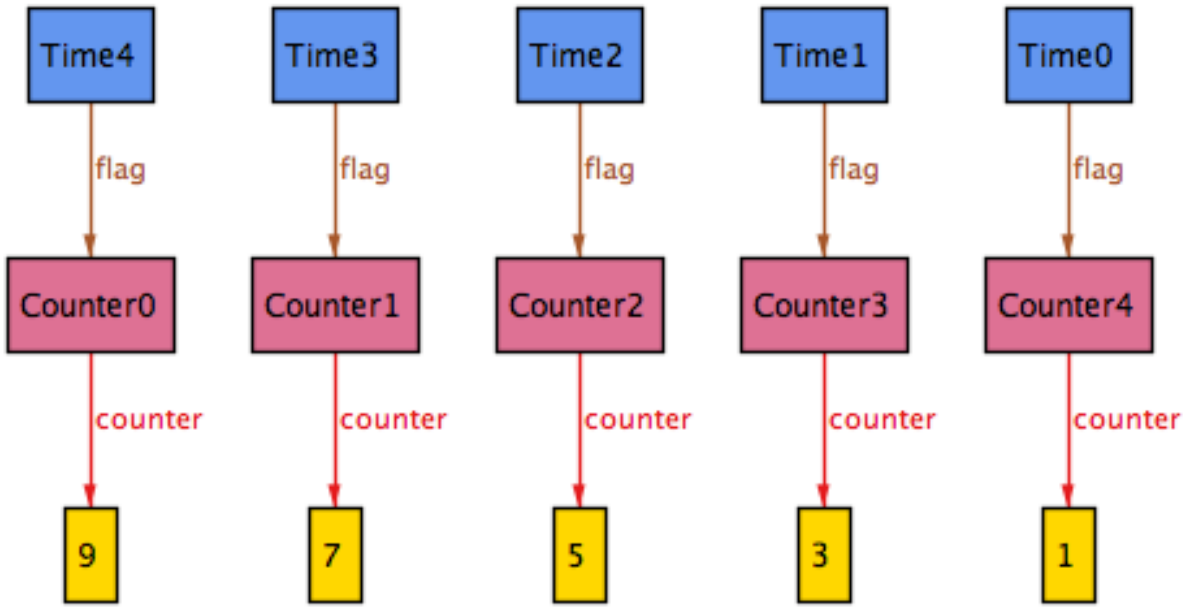

Figure 5.6: "Counter" incrementation throw Instant

5.3.1 Create a Formal Model

In order to create the model, it is essential to extract some information from the documentation for the system requirements, such as the states and the transitions between the states. The tool is designed to be interactive with the user and to work side by side with the Alloy Analyzer in a more iterative way. The user can execute the model since the very beginning and return to the tool to make some update or changes in the model.

The key functionality of this tool is to create a model capable of simulating a state machine. As shown in Figure 5.7 the application stores the inserted new states in a visible list on the left side. In the right side appears the list of the inserted events. These two lists work together to make a valid transition. In the middle area the creation of the transitions is presented. The user chooses the origin and destination states from the available states list and the needed possible event for this transition to occur. The new transition is presented in the list at the bottom. All these inserts and new signatures are controlled and verified by the tool to reduce human error in the modelling phase, resulting in a more reliable and fast way to create an Alloy model.

The upper bar in the tool gives the user the event to load a previous ModelMaker project and to save the current model in a new project. Figure 5.8 shows the addition of a new `State` and of a new `Event`. The application just requires a name to create states, but, when adding an event, the user has to choose if it is an abstract signature, and always

Figure 5.7: ModelMaker Interface

choose a a previously defined event to extends this new event. This definition (always a `Event` sub-class) is crucial to create an hierarchy for model checking proposes. In case of a new previously defined event, the standard `Event` is added as extended class. Both functionalities allow the insertion of multiple objects before returning to the main program.

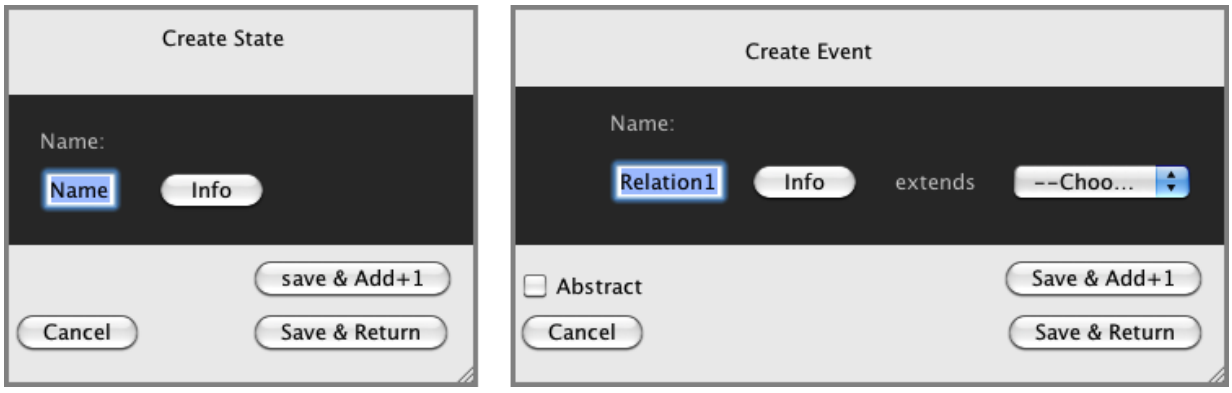

Figure 5.8: State and Event addition to the model in ModelMaker

If the user decides to generate the model at this point, the window in Figure 5.9 is shown, giving the opportunity to choose some more model characteristics: for example,

the maximum number of transitions that model ensures; invariants and the checking restrictions described for the model. Next, the starting state is selected, this is the first state in the model. At last the user can choose a name for the output Alloy file, that by default is the same name that the user chose to save the ModelMaker Project. He can always return to the main menu to add some more model details and generate the model again. Doing this iteratively the model grows as more and more information is added to the project.

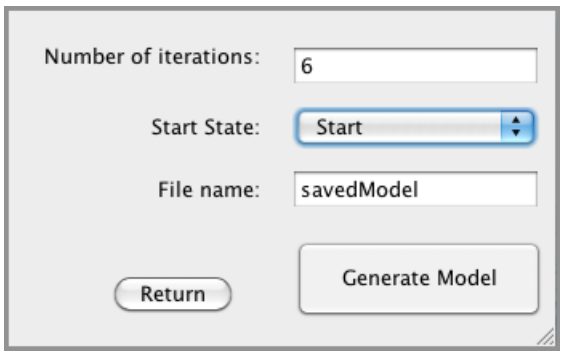

Figure 5.9: A new model creation in ModelMaker

5.3.2 Checking the Model

Assertions and facts are used in Alloy to verify and check model properties. When the model is created this application gives the user the opportunity to edit and insert more information. The user can give and define more information to the model invariant and check if the new alterations and properties hold for a new iteration in the model simulation.

By using this, project's important critical properties and restrictions are ensured during a real execution.The model guidelines used to create the model schema, described in subsection 5.2.2, have demonstrated, in these models creation, to be time saving and useful in reducing the complexity of the initial steps of the process.

ModelMaker supports the creation of standard `asserts` such as loop verification, self loops and starvation properties with a simple bullet selection in the application interface. These implemented functionalities give the user a start-up environment to check state properties such as: verifying if the model contains loops; if a state is self-reachable or if a state can get to a possible undesirable situation.

ModelMaker provides a more friendly way to create a system's first formal model in Alloy. With only a few steps such as: inserting the model transitions; system validations and invariants and simulation attributes, the application returns as output the Alloy file, as in Figure 5.10.

Now, the user just has to execute the model (1) and the generated instance of the model

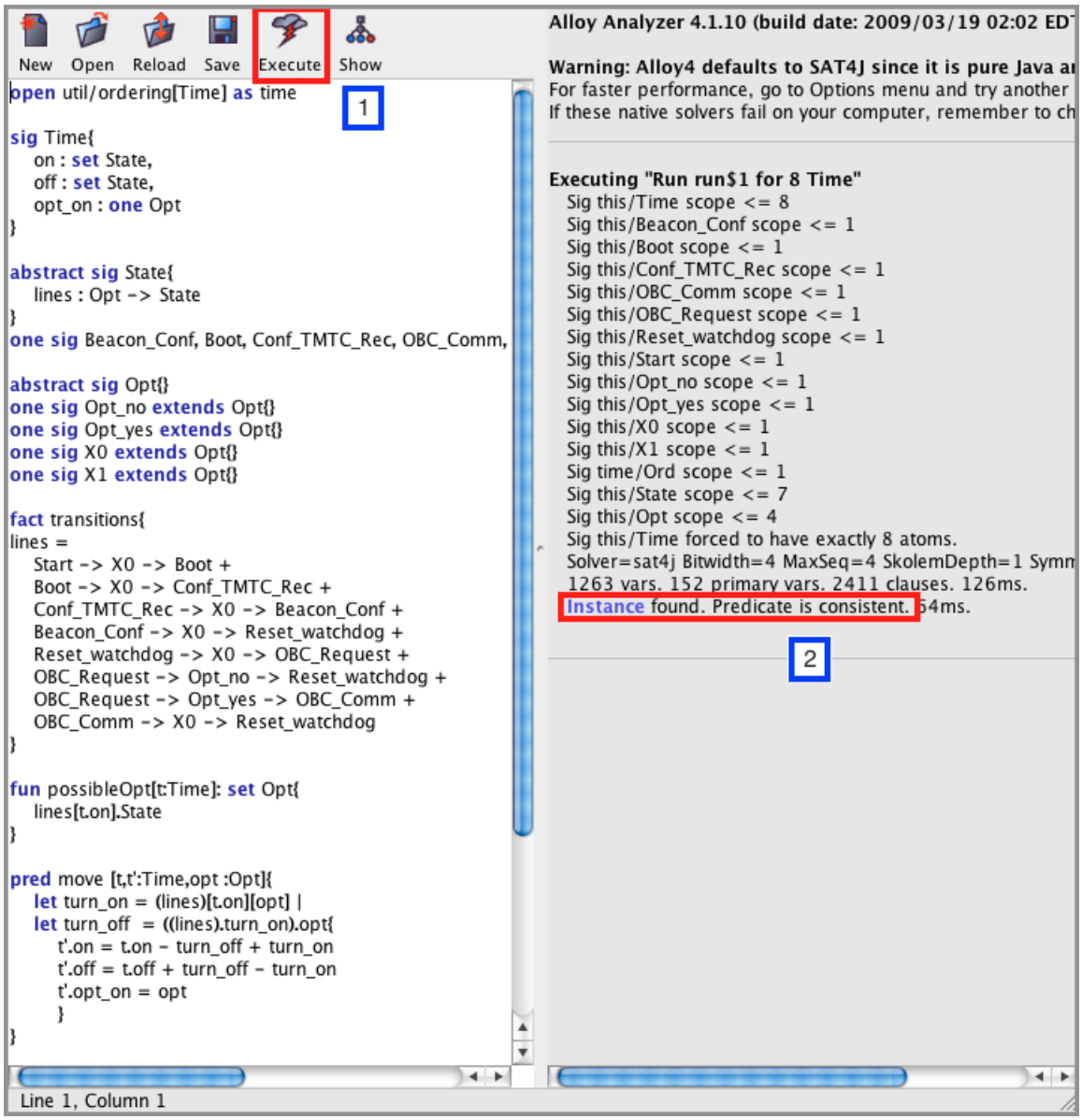

Figure 5.10: Alloy file resulting as a ModelMaker output

(2) is ready to be seen and analyzed.

MM $\Longleftrightarrow$ Alloy

The process of adding new features in the application and viewing the result model instantaneously in Alloy Analyzer can be seen as a iterative layer-growing procedure, that gives more and more value to the model in every iteration.

5.3.3 ModelMaker Implementation

The ModelMaker application was created in Java and can run in every operating systems that allows Java programs. Since Alloy also has the same requirements, they make a perfect match. However, ModelMaker implements features that help the (MM,Java) integration and configure the system's directories.

Every time that a change in the MM project is made, the user can reload the Alloy file in the Alloy Analyzer and the modifications are applied automatically, allowing a sense of iterative building of the model.
A model of the ModelMaker application was made using the application itself for controlling and checking requirements, see Figure 5.11.

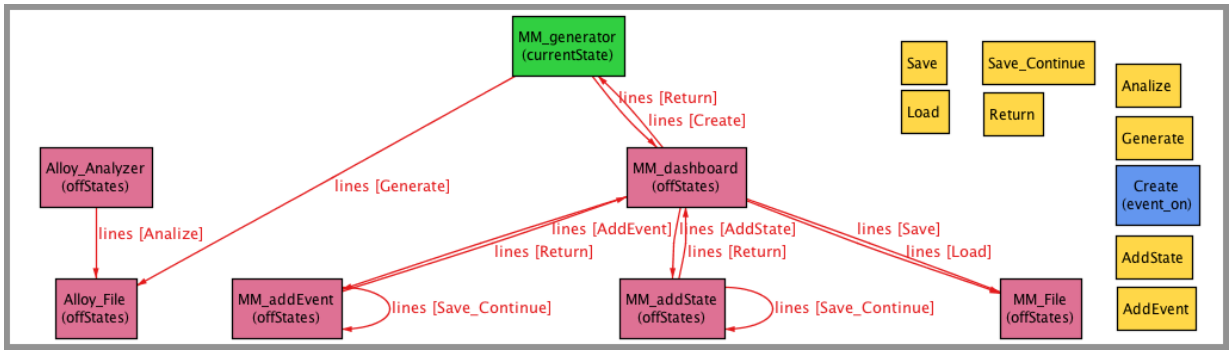

Figure 5.11: ModelMaker model created by ModelMaker tool

5.4 Conclusion

In Section 5.2 we reduce model creation time by giving some guidelines as a starting point for the system model. The developer can make a formal model capable of checking invariants and restrictions defined in the project requirements. This template has some functionalities such as: simulating the system flow, verifying the behaviour when adding synchronization and exceptions, and working with Booleans, Integers and Flags.

Furthermore, we implemented a prototype that can generate quickly and accurately the described model template - the ModelMaker application. This application aims at reducing even more the model creation time and generating a model in Alloy. The user can have the initial formal model project created and ready to be analyzed with Alloy Analyzer tool; furthermore, the model can evolve and grow by applying more features in ModelMaker or simply adding them manually in the Alloy file.

This tool can be very handy to start a formal design in the very beginning of the project. It can create the bridge between the user and a formal model with just a few

steps. At a starting point, the user can inject some automatic rules and then improve the model defining some other pertinent requirements.

The tool was design to automatic generate models using the guidelines previously defined in Section 5.2 and was created to support only models that can be described as state machines.

"First, if you use models during design, you catch errors early, when everything is still small. And, second, in hindsight, most software design issues are trivial.", [Jackson, 2012].

Chapter 6

A Real Case Study!

6.1 Introduction

In this Chapter, the previously defined methodologies are applied to the ITASAT project. The overall coordination of the project in made by the Brazilian Space Agency program for developing and launch satellites, while the responsibility for the project execution is assign to the Aeronautics Technological Institute. This project is being developed by graduated students from several engineering areas. The aim of this work is to support the ITASAT project in some project's phase such as: planning, design, code with validation and verification procedures. This work goes through the early conception of the "Satellite System Modules", the "Software Modules","Communication Modules" until a more low level modelling such as the "Receiver Code".

In order to do this work, we use the ModelMaker tool to create the first formal model in Alloy in a more easy and quick way. All processes were evolved by updating the MM project or adding restrictions and necessary validations directly in the Alloy file.

6.2 Satellite System Modules

This Section introduces the work made in the ITASAT project, we start by verifying the Satellite routines. The diagram of the System Modules, Figure 6.1, represents the expected behaviour of the satellite, since the satellite launching until his survival and maintenance in space. It also has a test module here the communications between the base on earth and the satellite in space are simulated through a cable called "Umbilical Cable".

The diagram was built following the requirements in the documentation. Our goal was to validate this diagram against those requirements and verify is the diagram was trustful. In order to accomplish that, a formal model was created resorting to ModelMaker and to some the guidelines defined in Section 5.2.

As shown in Figure 6.1 the system is composed by seven modules: `AIT`, `Lauch`, `Survival`, `Idle`, `Alignment`, `Alignment Payload` and `Payload`, and necessary connections to communicate with other modules: `Umbilical`, `Telecommand`, `Automatic Transition` and

`Failure`. In the following Sections found errors are described.

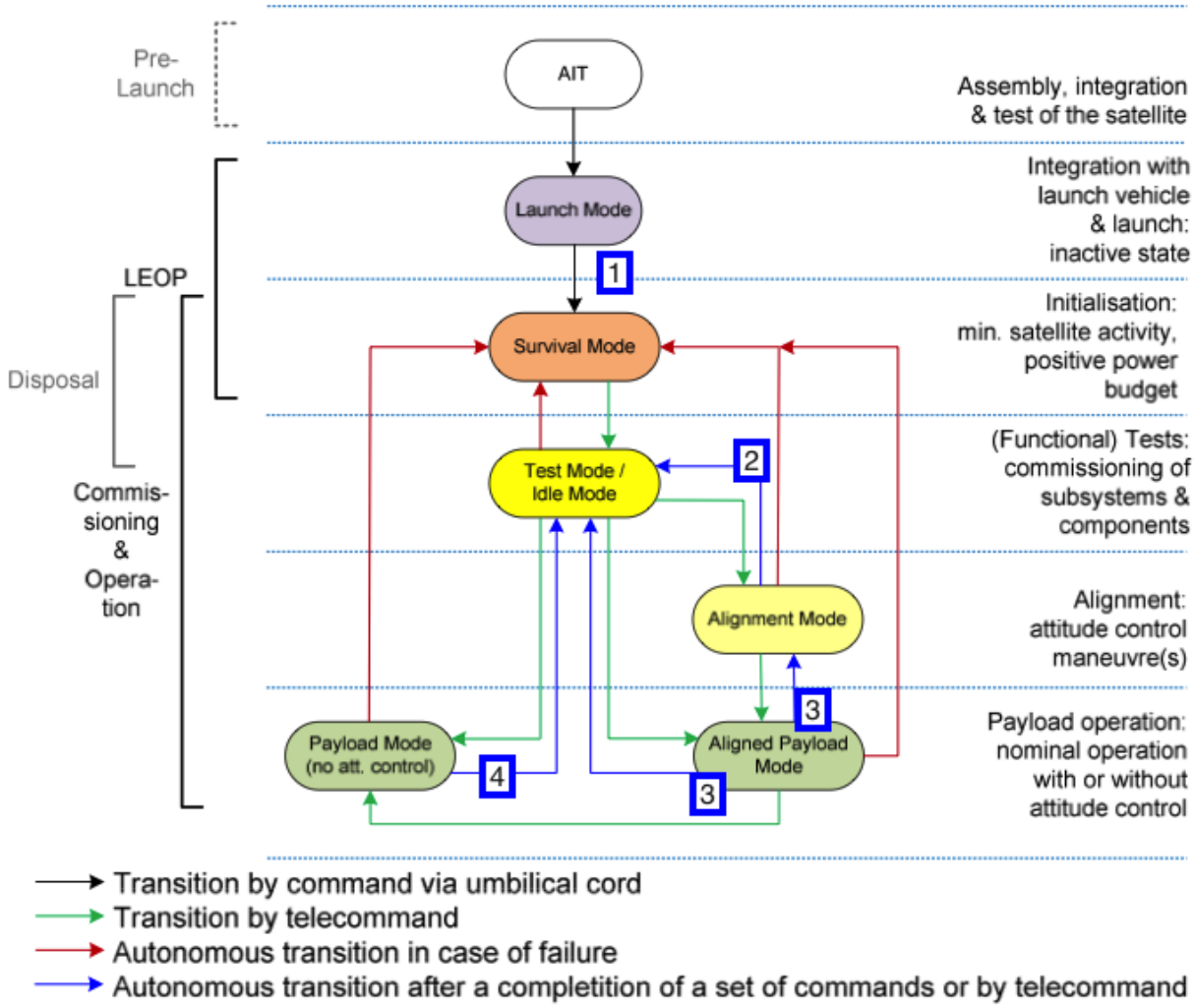

Figure 6.1: Satellite modules ITA documentation

6.2.1 Unreachable Survival Mode

The transition between `Launch Mode` and `Survival Mode` is simulated by umbilical communication, still this physical communication cease to exist and is no longer valid after the real launching. An automated transition should be defined that makes available the transition between this two modules. This fault was found in Alloy by removing the transitions that are no longer valid in space, the Umbilical transition:

```
...let initInstant =  LaunchMode|...

fun possibleEvent[t:Instant]: set Event{
    lines[t.on].State - earthEvent[t]
}

fun earthEvent[t:Instant]: set Event{
    Umbilical
}
```

Without the earth transition, the satellite became obsolete and becomes a piece of metal without the turn-on functionality and the satellite would never had the possibility to experience is live in space.

6.2.2 Starvation in Alignment Payload Mode

Alignment Payload is a mode that can be reached from *a*)`Alignment Mode` and *b*)`Idle Mode`. `Alignment Mode` has an automatic Mode transition that does not need any external interaction to exit and return to `Idle Mode`, therefore the transition `Alignment Mode` → `Alignment Payload Mode` tends to be inefficient. There is a direct transition b) Idle Mode → `Alignment Payload Mode` that has the same result and can substituted *a*).

This check was made by an assert in Alloy that verified that a transition that needs an external interaction in a module cannot be made in a state that has an automatic valid exit. Figure 6.2 shows some transitions that should be avoid for a more robust system.

In critical systems jumping around in secondary states should be discouraged. So, in this situation, the procedure should be returning to the main and more robust module (`Idle Mode` or `Survival Mode`) and from there directly to secondary and more complex modules.

6.2.3 Duality in Alignment Payload Mode

In Figure 6.1-tag 3, it is possible to see two similar transitions exiting from `Alignment Payload Mode` and both from automatic transitions. This situation does not hold the invariant of the model that assures that in any moment the model can have just one `CurrentState`.

The Alloy Analyzer found a state that does not hold the invariant and model restrictions. Figure 6.3 shows that in this situation the model can evolve to an invalid state were it is possible to have more than one state in use. This situation violates the invariant: `pred invOneCurrentState[t:Instant] {one t.on }`. The decision in `Alignment`

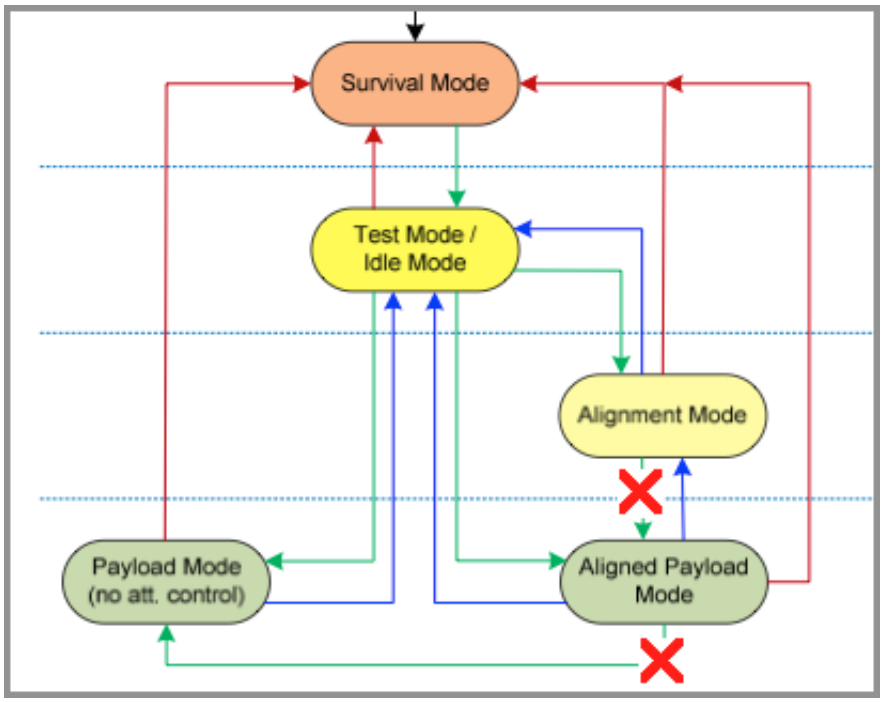

Figure 6.2: Simplification in Satellite System Modules ITA

`Payload Mode` should be decidable and not ambiguous transitions with an automatic transition.

6.2.4 Payload Mode Smooth Exit

The model found that the state "Payload Mode" fail when it tries to validate the assertion `notSmoothExit` that validates if every state can transit to another state without using the "Failure" transition. The assertion is described as:

```
fun possibleEvent[t:Instant]: set Event{
    lines[t.on].State - notSmoothExit[t]
}

fun notSmoothExit[t:Instant]: set Event{
    Failure
}

assert smoothExit{
   all t ,t':Instant | some event : possibleEvent[t] |
```

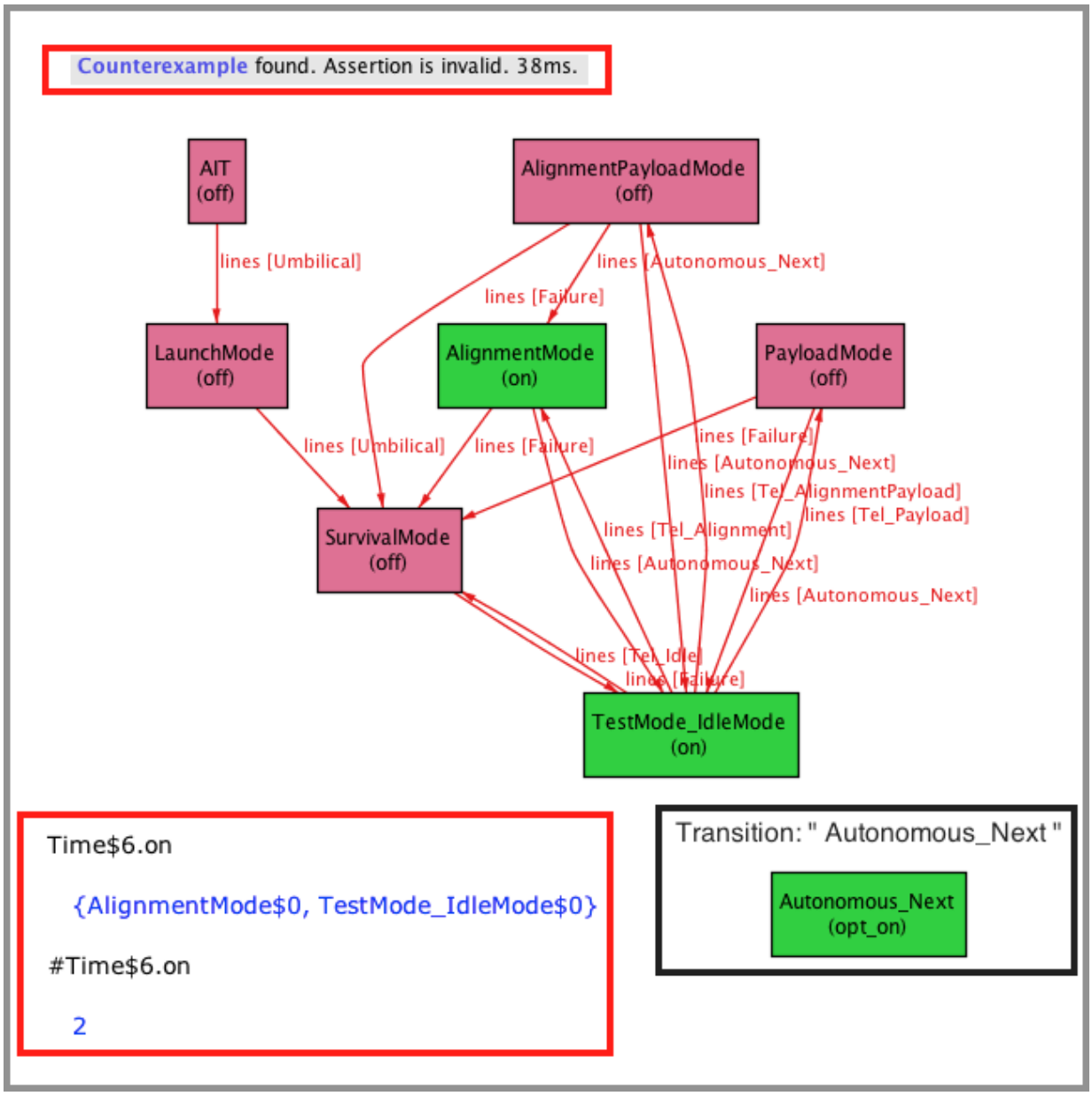

Figure 6.3: Alloy Analyzer found a invalid `Instant` that violates the invariant(one `CurrentState`)

```
    invariant[t] && move[t, t', event]  => invariant[t']
}
```

The transition in Figure 6.1-tag 4 was added after the fault verification. This transition was later added to the model. Now it is possible to retrieve to the `Idle Mode` by an automatic transition or by telecommand sent by earth.
The fault found was that from a reachable state it is not possible to transit to another state without using the `Failure` event. This shows that there is a final mode in the system that recovers only by failing. And in the case of the "Payload Mode" there should be an extra predictable transition to a more robust state. This transition was added to the model as seen in Figure 6.1. Now it is possible to retrieve to the `Idle Mode` by an automatic transition or by telecommand sent by earth.

6.3 Software Modules

This Section presents the work made in the Attitude Control and Data Handling (ACDH) software Modules. The system starts after a mechanical separation of the satellite from the launcher, that triggers the watchdog for starting the electronic devices on Instant. The modules are: `ACDH Power Off`, `CM Inicialization`, `CM Survival`, `OBC Inicialization`, `OBC Sequential` and `OBC Nominal`, those are the module, however three more test modules where added: `CM Test`, `OBC Initialization` and `OBC Sequencial Test`.

6.3.1 Unreachable CM Initialization

As shown in Figure 6.4, `CM Initialization` is only reached by an `Umbilical` transition, represented by the black arrows. However, when the satellite is in space the umbilical cord communication is ensured by earth telecommands and in this case the module `CM Initialization` it is never reached, this problem is an example of "error propagation". The problem was found in the "Satellite System Modules"- Figure 6.1-tag 1, diagram and propagated to a lower level in the specification, which means that if the fault was previously discovered, this error could have been fixed. This fault is shown by checking the `check notInEarth`:

```
fun possibleEvent[t:Instant]: set Event{
    lines[t.on].State - earthEvent[t]
}

fact init{
    let t0 = Instant/first |
        let initInstant =  AIT |
            t0.on = initInstant && t0.off = State - initInstant
}
```

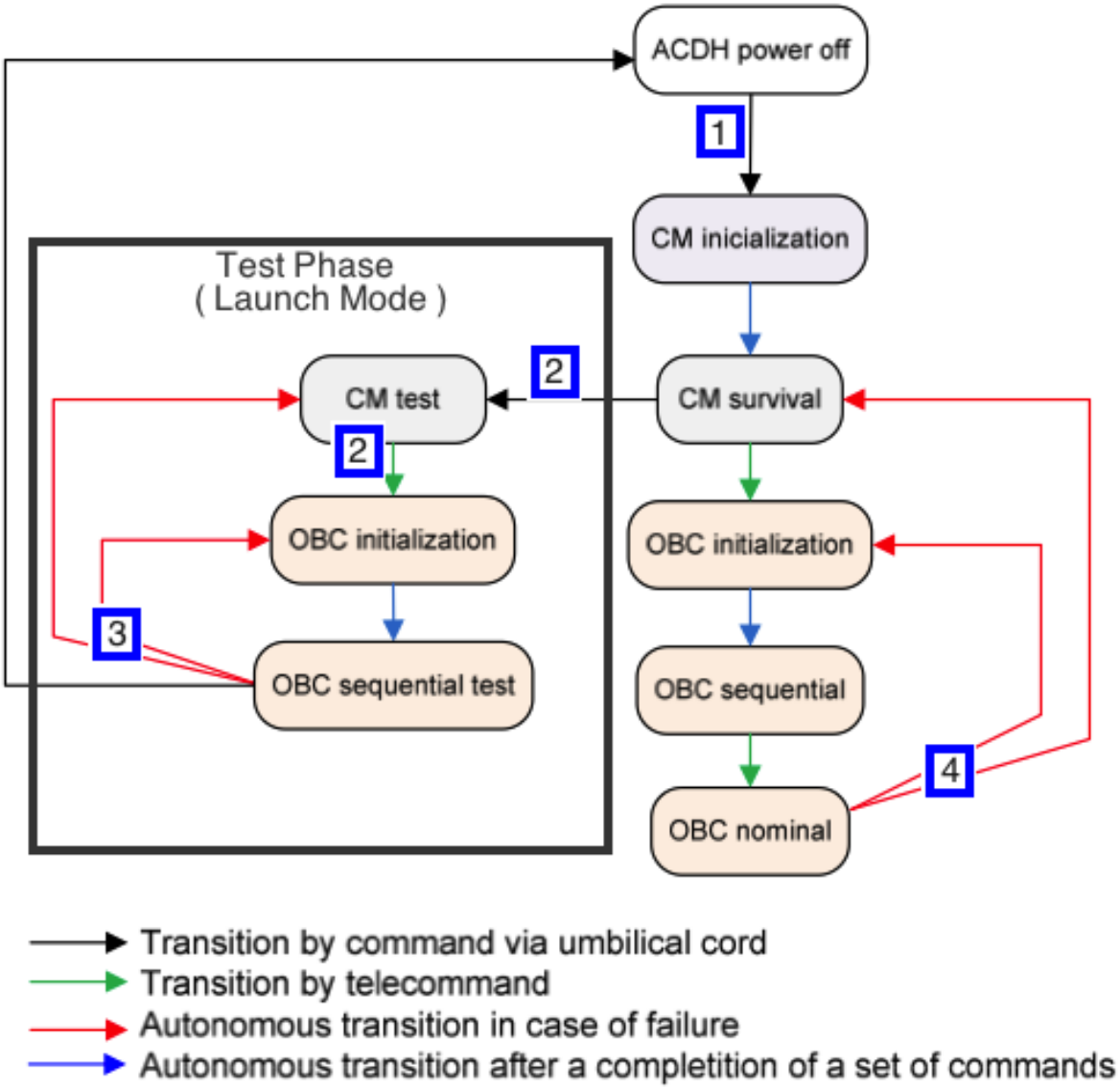

Figure 6.4: System modules ITA documentation

This fault is a propagation error that derives from the previous model. When the satellites modules were defined, the available state after the launching was also unreachable for the same reason. This is a remarkable case were a small mistake can evolve to a important issue and potential problem when defining a Satellite that cannot be repair

after is sent to the space. This starting phase is one of the more critical parts in a Satellite life-cycle, because without the first boot the satellite is forever useless.

6.3.2 Incompatibilities in Communication

The Test Phase in the software model seen in Figure 6.4 shows that this phase is accessed only by a command sent by the umbilical cord. Furthermore, `OBC Initialization Test` is achieved only by earth telecommand. This type of transition is not possible in Test Phase because is not physically possible to receive commands at the same time by umbilical cord and receiving telecommands by earth. This fault was found by Alloy Analyzer by checking the reachability of all test modules on earth and the other normal flow modules in space.

```
fun possibleEvent[t:Instant]: set Event{
    lines[t.on].State - earthEvent[] //comment OR spaceEvent[]
}

fact instantTransition{
    all t :Instant , t' :instant/next[t] |
        some op : possibleEvent[t] | move[t,t',op]
}
```

As seen in the code above, the possible events are reduced to the events in space or the events that the satellite have on earth by subtracting `earthEvent` or `spaceEvent` respectively.

6.3.3 OBC Sequential Output

The requirements of the software module define a test simulation when the satellite is assembled in the launcher and ready to go to space. This test uses the umbilical cable for simulating the communication sent by earth to the satellite in space. However, the modules that are going to be tested are the same modules in the real system flow. In Figure 6.4 is possible to see that the `OBC Sequential Test` has two failure transitions that are not represented and defined in the `OBC Sequential`.

For the same module more privileges are given when it is on the testing phase. This type of advantage should not exist because it gives a false security that cannot be resolved in a real case flow and causes an uncontrolled state.

6.3.4 OBC Nominal Smooth Exit

Similar to the problem is Section 6.2.4, this model fails when checking the `smoothExit` predicate. This predicate verifies if all states can transit to a adjacent state without the need of a Failure transition. In this model `OBC Nominal` module can just return to a more robust module when an error or fault occurs. A autonomous or telecommand transition should be added to increase the control and maintenance of the system.

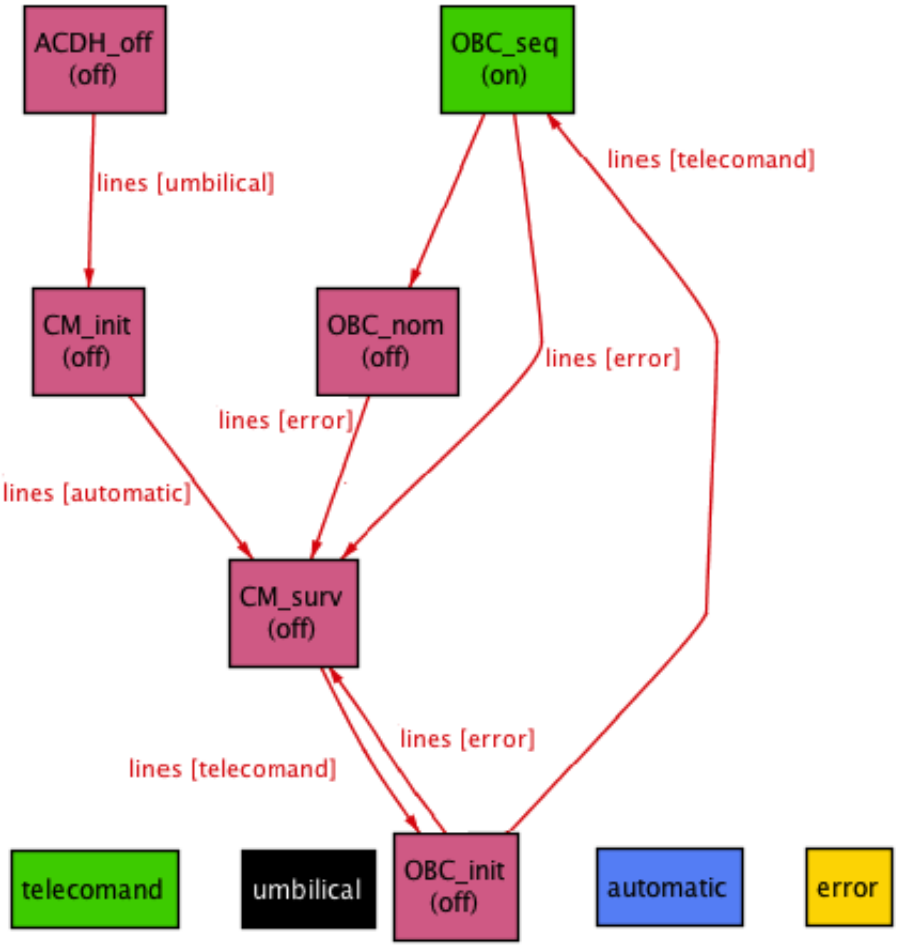

Figure 6.5: Software modules alloy

6.3.5 Inconsistency between Diagrams and Requirements

The objective of these models is to verify the consistency off the explicit requirements in the documentation in the project several phases. Although, there are different models that correspond to the same situation in the satellite life, so, for this reason is imperative that they can be synchronized from the model in design phase to the program in the code

phase. Table 6.3.5 shows the involvement between the `Satellite Modules` and `Software Modules`.

a) Satellite Modules	b) Software Modules
AIT	ACDH Power Off
Launch mode	ACDH Power Off CM Initialization and CM Test OBC Initialization test and OBC Sequential Test
Survival mode	CM Initialization and CM Survival
Test mode / Idle mode	OBC Initialization and OBC Sequential
Payload Mode Alignment Mode Alignment Payload Mode	OBC Nominal

Table 6.1: Alignment between Satellite Modules and Software Modules

However, if a comparison is made between both diagrams, `Satellite Modules` - Fig 6.1 and `Software Modules` - Figure 6.4, we find that are several differences that should not exist. Figure 6.6 shows several incompatibilities.

The transition in these two models should have been isomorphic, however some differences were found:

- In diagram a), `Test Mode` has a transition (Failure) to `Survival Mode`, and in diagram b) there is not a transition between `OBC Initialization` or `CM Survival` to `CM Initialization` or `CM Survival`.
- In diagram a), `Alignment and Payload Modes` have a autonomous transition to `Idle Mode`, and as shown in the diagram b) there is not a any transition from `OBC Nominal` except by failure.
- In diagram b), `OBC Nominal` has a transition (Failure) to `OBC Initialization`, but in the similar a) diagram all Failure transitions have `Survival Mode` as destination and secure model.
- In diagram b), `Test Phase` ends with a transition (Umbilical) to `Power Off`, in diagram a) the model cannot even reach the Test Mode because an autonomous transition is needed to access `Test Mode` from `Survival Mode`, however, if that transition exists there is not a transition to `Launch Mode` again to finish the testing phase.

6.4 Communication Modules - Receiver

The CM Receiver Module has two receivers, one Nominal and other Redundant and it is inserted in the ACDH Module in Satellite sub-systems. The receiver is very important

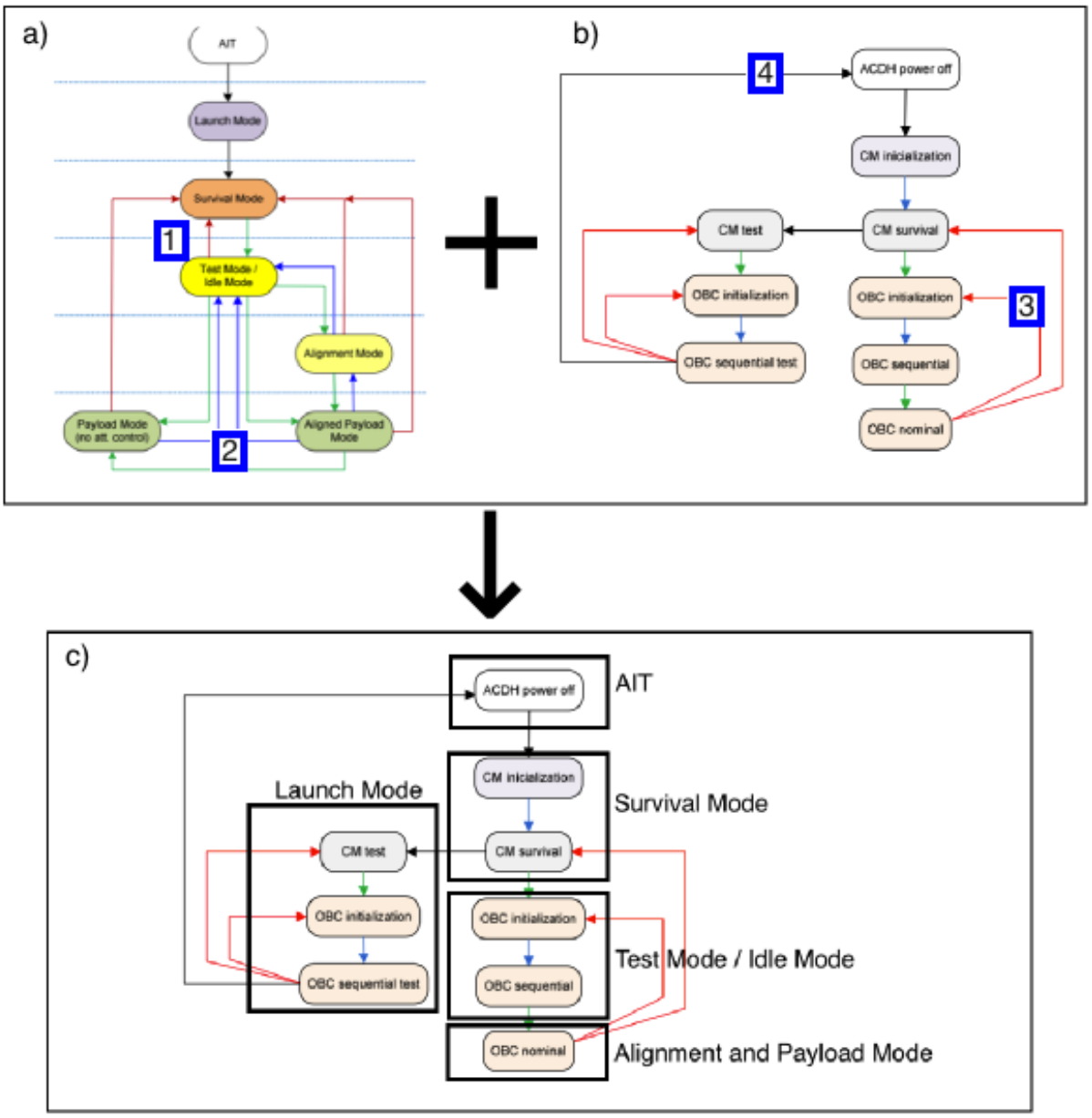

Figure 6.6: Overlap of Software Modules by Satellite Modules

in the communication between the satellite and Earth and it is responsible for several vital activities such as:

- perform a memory check during the booting stage;
- verification of OBC request signal. It shall be performed continuously while the CM receiver is in the active mode of operation;

- receive telecommands sent by Earth or commands sent by OBC.

6.4.1 Foreground Routine - Receiver

The CM Receiver is divided into two receivers, Figure 6.7 shows the foreground routine of the Nominal Receiver. This critical modules should be carefully verified because the receiver has the ability to shut down the entire system.

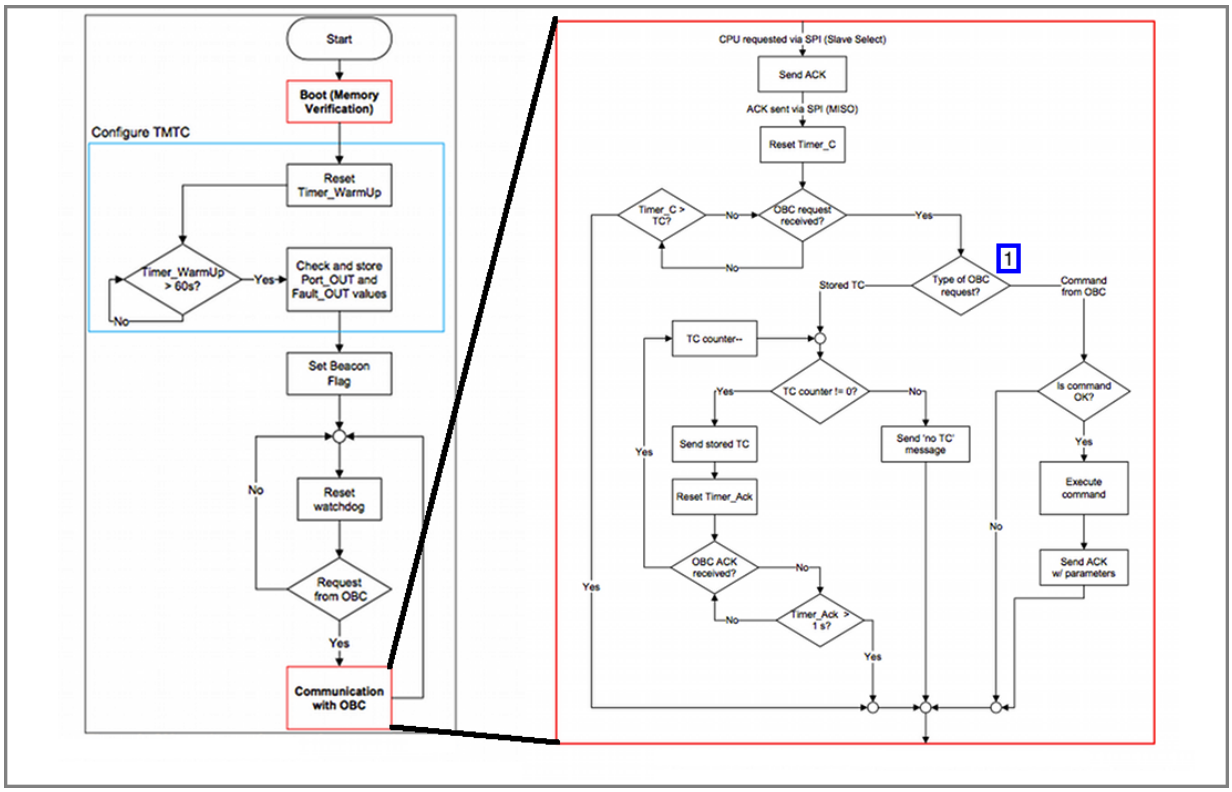

Figure 6.7: Receiver, Communication with OBC diagram [ITASAT documentation]

The extended diagram of the "Communication with OBC" represents the communication with the processor and memory in OBC. This communication has two different goals: Executing commands sent by OBC or storing Telecommands (TC) in OBC memory and in tag[1]-Figure 6.7 the system verifies what type of request it is going to receive: a `Stored TC` or a `OBC Command`, however there is a third option, a `TC Error`, that is not checked in this system.

6.4.2 Survival Flag Interruption

The `Survival Flag` in one of the three `Interruptions` that can interfere with the `CM Receiver software`. This modelling is made using the guideline defined previously in Sec-

tion 5.2.6. In order to model interruptions, it was created a second `Instant` flow `abstract sig interruption1 extends Instant` and verified in every iteration if the randomly generated boolean flag is activate to trigger the interruption shown in the code below and in the Figure 6.8.

```
pred moveM[t, t' : Instant, event, eventE : Event]{
    t.bool = True implies{
        t.onE = Exit implies{
            t'.onE = Interruption
            t'.offE = modelo2 - Interruption
            t'.event_on = none
            t'.bool = False
        }else (...)
```

In this code the two verifications are shown, the boolean flag and the exit point in the interruption informing that the interruption ended.

Due to the responsibility of this module in all the satellite communication processes and to return to the `Survival Mode`, further validation and verification was made. An important fault was discovered when sending the Survival Mode Interruption. The Survival Mode is defined in the Transceiver Module to have this configuration: Receivers in On mode and the Nominal Transmitter in Beacon Mode as the Nominal Transmitter in Off Mode as shown in Figure 6.9.

The Nominal Receiver has the functionality to turn the Nominal Transmitter in Beacon-Mode, however, the Redundant Receiver also has to do a similar function to the Redundant Transmitter but there is not a direct communication between the two off them. This situation provokes that both Receivers act as **Nominal** and turn both Receivers to `Beacon Mode` and have both transmitters transmitting is a critical fault.

During the modelling process in the "Receiver communication with OBC" and the survival flag. A major fault was found. The transition between the operation modes of the CM transmitters (receiver and transmitter pair). It is written in the specification that if a "Shut Down"command is received the `Receiver` can turn off the main module and should also give instructions to its transmitter to change its mode to "Survival Flag". However, in order to have this sequences of events a link between the pair (receiver,transmitter) should exist. After evaluating this problem with the mechanical group, it was realized that it was not possible to make this physical implementation and the previous processor had to be revalued and changed. This modification had a strong economical impact in the overall project. Still, this change could become much more difficult and expensive in a more advanced project phase.

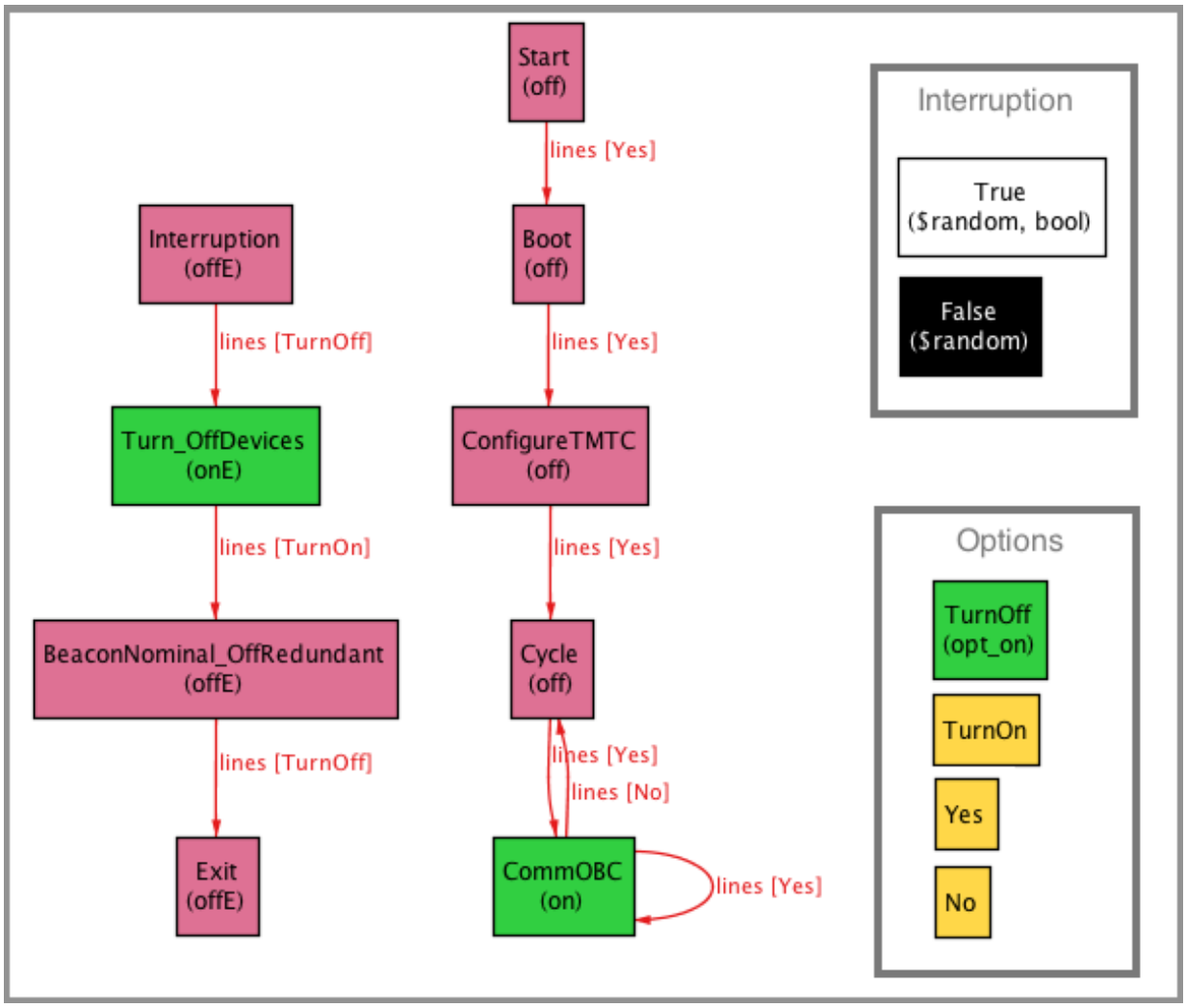

Figure 6.8: Communication with OBC, Interruption Alloy Model

6.4.3 Memory Read and Write Pointers

It was made a more deep consideration about the memory to verify it in terms of usage. It was defined a model with just four memory slots to simulate the real one in the satellite. It was defined the predicates `readAction` and `writeAction` to change the main state, `read` and `write` as pointers that represent the next slot to read or to write and `full` and `empty` as slot attributes.

```
sig State{
   read : one Slot, -- read pointer
```

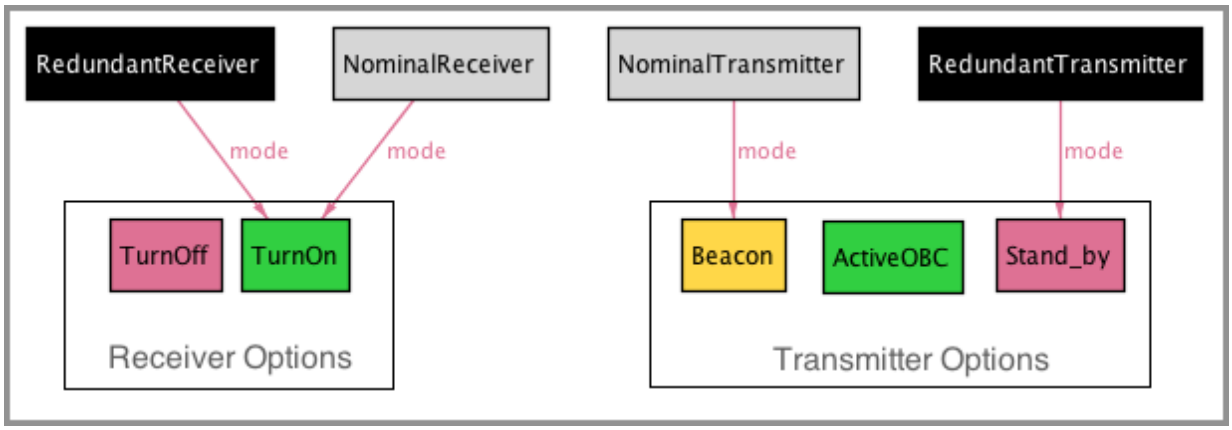

Figure 6.9: Communication with OBC, Interruption Alloy Model

```
   write : one Slot, -- write pointer
   empty : set Slot,
   full : set Slot
}
```

```
pred readAction[s,s':State]{
   let next_slot = s.read.next{
      s'.read = next_slot
      s'.write = s.write
      s'.empty = s.empty+s.read
      s'.full = s.full - s.read
      }
}
```

```
pred writeAction[s,s' :State]{
   let next_slot = s.write.next{
      s'.read = s.read
      s'.write = next_slot
      s'.empty = s.empty - s.write
      s'.full = s.full + s.write
      }
}
```

We have to define the initial instant state that both pointers are pointing to slot1 and every slot in empty. In the end we make the condition to go to the next instant state, you read or you write.

```
fact{
   let s0 = state/first | s0.read = slot1 &&
      s0.write = slot1 && s0.empty = Slot && s0.full = none
```

```
   all s: State, s' : state/next[s] | write[s,s'] or read[s,s']
}
```

We define two assertions, one to test reading an empty slot and by opposite writing in a full slot. As we can see, in tag[1]-Figure 6.10 the pointer is in slot1 and the action is to read from it. In this case we will have a problem and a dead-end, the programme will try to read an empty slot and that may cause a programme failure, nothing is told to the programme about what to do in this situation. On the other hand when we try to write in a space that is already full, tag[3]-Figure 6.10, problems were also found, be stuck because the programme is not allowed to write, we can rewrite and lose the previous information ant it can mix the information and lead to a wrong instruction.

```
assert test_read{
   all s:State | s.read not in s.empty}
assert test_write{
   all s:State | s.write not in s.full}
```

The pointer actions are not fully specified. It is only defined that the first slot to read is the older one, so in this case, the pointer can read the first slot even if this was already written.

6.5 Conclusion

This work have a lot of importance not only in the software modules but also in the entire project development. A modelling was made from the system structural phase to the low-level receiver code and memories. In order to create these models, the ModelMaker tool was used and proved to have a great impact in terms of efficiency and to create a first initial model in a quick way.

This verification started from the Satellite System Modules, where several faults and ambiguities were found; then a verification in Software Modules was made and several mistakes were found; next, a comparison was made between these two models, with the expectation to find two isomorphic models. This assumption proved to be wrong and a more precise verification was made to check the differences that should not exist in those models transitions.

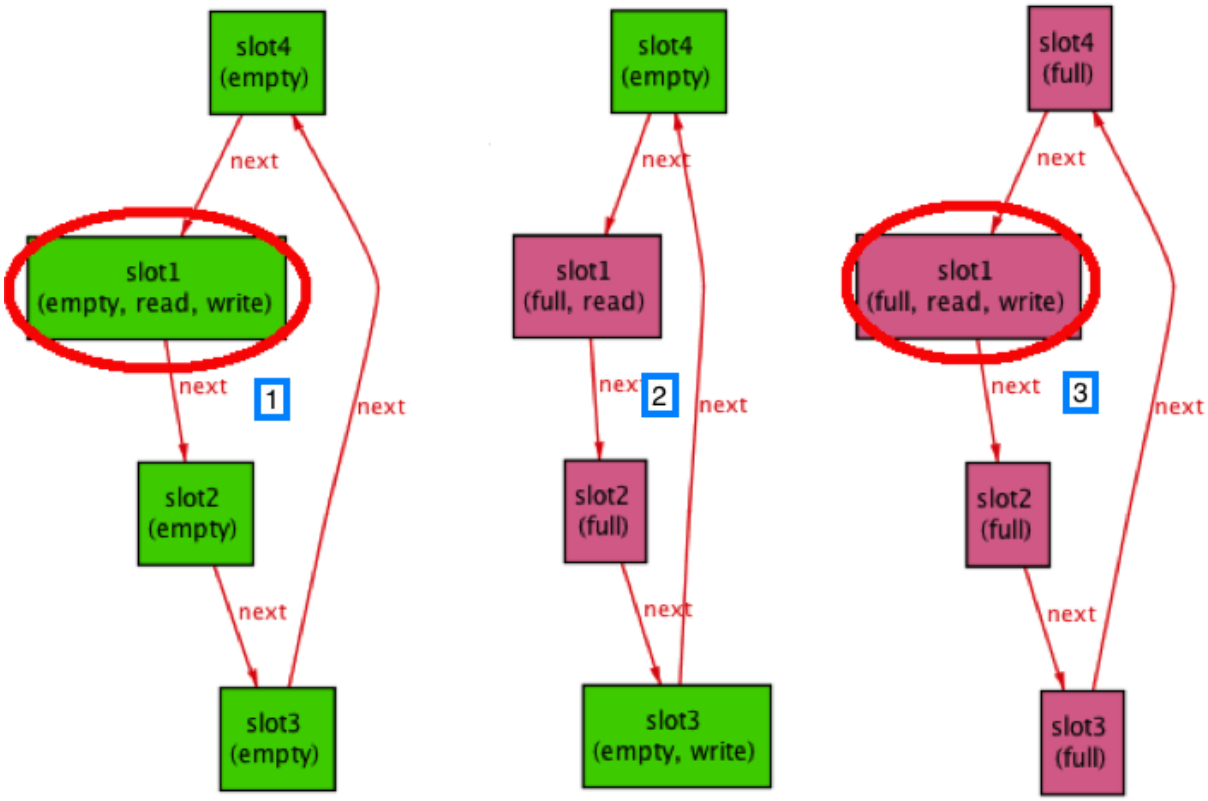

Figure 6.10: Three different states (1,2,3) of the Memory

Then, a formal verification was made in the Communication Model in order to check the requirements and the interruptions. This proved to be difficult and time consuming even with the ModelMaker tool supporting the analyses. In the end, an critical fault in the Receiver and Transmitter communication was made and discovered changes in the satellite processor and architecture had to be considered and a change in the satellite structure had to be made.

This project impact consisted in a total revaluation of the satellite project due to some incompatibilities and structural limitations that meant an extra expense that could have a worst impact in the project budget if found in a later phase.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This work presents an approach to modelling behaviour, systems and critical software that could be described as a state-machine model. The ESA report mentioned in Chapter 3 provided a helpful starting point in the development of the Validation and Verification methodology, since some Formal Methods techniques were introduced in order to give a more theoretical support to the process using Model Checking. The two NASA guidebooks, also mentioned in Chapter 3, are designed to introduce both project managers and software developers to basic formal methods concepts and some of the most important issues involved in deciding when and where the application of formal methods is advisable and cost effective. The introduction of formal methods in industry can provide evidences to support Hoare's vision of a future world in which computer software is always the most reliable component in any system which it controls, and no one can blame the software any more [Hoare, 2007].

The overall goal of the project was to provide a more linear and organized verification and validation plan. The aim was to formally verify the most critical parts of ITASAT project, supported by Alloy models and proceed side-by-side with a formal modelling in several project phases. Alloy and its model checker tool, the Alloy Analyzer, were chosen due to the combination of first-order logic and relational calculus in a language with a syntax for structured specifications. Moreover they combine singular characteristics: Models and Micromodels, Analysable Models, Declarative Models and Structural Models. Alloy enables building an easy initial model without having the complete specification, this model can evolve iteratively side-by-side with the new project requirements. The choice of Alloy was also justified by the possibility of comparing its behaviour with UPPAAL, a formal tool for specify and verify dynamic behaviour of projects, when applied in the critical technological satellite context. UPPAAL, in fact, was previously used by an ITASAT group member to model an embedded satellite system.

In order to formally model the system, the following steps were adopted: a) the requirements were manually extracted from the documentation: in this phase we faced difficulties

deriving from the lack of a full requirements specification document of the project and from the contradictions between the system and module diagrams and the correspondent requirements; b) the guidelines for an initial Alloy model creation and formal verification and validation were defined including some extra features such as synchronization (Section 5.2), interruptions and flags: in this phase we faced the challenge of transposing the textual requirements into the Alloy logic language; c) a tool, ModelMaker, was developed to automatically assist the user in the creation of the model: here, a representation of the Alloy models had to be designed in Java in order to create and store the models (Section 5.3).

The defined guidelines were then applied to a real case study (Chapter 6), the ITASAT project, concentrating on its use at the earlier stages of specification and design. We also made use of the ModelMaker, which helped generating the necessary models for the verification and validation process in the different work phases: from Architectural Design, Communication Module to Transceiver Code. This process is undoubtedly useful and time saving. However, the object of study should allow a state-machine abstraction to make the construction of the model possible.

Due to the interaction with the ModelMaker and the Alloy Analyzer, the effort of modelling can thus be amortized, not only over the analysis and design stages, but also adding new features and verifying the behaviour of the system in the later stages of software development. The approach used allows detecting errors and glitches in the early phases of the project design, which were less complex to fix, since their propagation was avoided and later costs reduced. The visual representation of the models in Alloy can be used as state diagrams in the specification documents: being formally created and validated, the diagrams gain great reliability. This feature solves the previously highlighted problem of the documents diagrams inconsistency.

Finally, we would like to point out that the existence of guidelines and their use in an automated tool may help convincing developers to start gathering requirements and built the formal model, first automatically and then manually translating those requirements to the Alloy model. With this approach, the user receives a visual model in a short period of time. The model is created using the specification and the specification can be improved with the model, making a symbiotic relation.

7.2 Future Work

These guidelines could be the beginning of a more complex and robust methodology to build a formal model. However, this set can be expanded and improved, adding consequently more features to the ModelMaker tool. As future work, it is intended to apply the guidelines to other systems, given the opportunity to introduce formal methods in the industry and to make them a more used approach in the improvement of the projects.

Bibliography

Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from uml to alloy. *Software & Systems Modeling*, 9(1):69–86, 2010. ISSN 1619-1366. doi: 10.1007/s10270-008-0110-3. URL `http://dx.doi.org/10.1007/s10270-008-0110-3`.

ANSI, SC, and ASQC. Quality system terminology, 1995. Quality system terminology.

T. Ball, V. Levin, and S.K. Rajamani. A decade of software model checking with slam. *Communications of the ACM*, 54(7):68–76, 2011.

B. Beizer and Van Nostrand Reinhold. Software testing techniques. Technical report, Carnegie Mellon University, 1983.

Stephen Bera. Structuring for the vdm specification language. In *Proceedings of the 2nd VDM-Europe Symposium on VDM—The Way Ahead*, pages 2–25, New York, NY, USA, 1988. Springer-Verlag New York, Inc. ISBN 0-387-50214-9. URL `http://dl.acm.org/citation.cfm?id=52257.52258`.

R.W. Butler, J.L. Caldwell, V.A. Carreno, C.M. Holloway, P.S. Miner, and B.L. Di Vito. Nasa langley's research and technology-transfer program in formal methods. In *Computer Assurance, 1995. COMPASS'95.'Systems Integrity, Software Safety and Process Security'. Proceedings of the Tenth Annual Conference on*, pages 135–149. IEEE, 1995.

CDRH. General principles of software validation; final guidance for industry and fda staff, 2002. food and drug administration.

Michelle Love Crane, Copyright Michelle, and Love Crane. Runtime conformance checking of objects using alloy. In *Electronic Notes in Theoretical Computer Science*, page 2003. Elsevier, 2003.

Duda Falcao. brazilianspace.blogspot.com, December 2011. URL `http://brazilianspace_blogspot_comaeb_pretende_lancarsatelite_itasat.html`. blog do ITASAT.

J.F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. Van Weerdenburg. The formal specification language mcrl2. *Methods for Modelling Software Systems (MMOSS)*, 6351: 34, 2007.

Hinchey and Bowen. Ten commandments of formal methods... ten years later. *Computer*, 18:40–49, 2006.

T. Hoare. The ideal of program correctness. *The Computer Journal*, 50(3):254–260, 2007.

Daniel Jackson. Structuring z specifications with views. *ACM Trans. Softw. Eng. Methodol.*, 4(4):365–389, October 1995. ISSN 1049-331X. doi: 10.1145/226241.226249. URL `http://doi.acm.org/10.1145/226241.226249`.

Daniel Jackson. Automating first-order relational logic. *SIGSOFT Softw. Eng. Notes*, 25(6):130–139, November 2000a. ISSN 0163-5948. doi: 10.1145/357474.355063. URL `http://doi.acm.org/10.1145/357474.355063`.

Daniel Jackson. Enforcing design constraints with object logic. *Static Analysis*, pages 95–102, 2000b.

Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11:256–290, April 2002. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/505145.505149. URL `http://doi.acm.org/10.1145/505145.505149`.

Daniel Jackson. Software abstractions-logic, language, and analysis, revised edition, 2012.

M. Jones and U. Mortensen. *Guide to Software Verification and Validation*. ESA (European Space Agency / Agence Spatiale Europenne), esa publications division edition, 1995.

M. Kaufmann and J.S. Moore. An industrial strength theorem prover for a logic based on common lisp. *Software Engineering, IEEE Transactions on*, 23(4):203–213, 1997.

J.C. Kelly. *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems Volume II: A Practitioner's Companion*. NASA, 1997.

J.C. Kelly, T. Lead, K. Kemp, and WV Fairmont. *Formal Methods Specification and Verification Guidebook for Software and Computer Systems Volume I: Planning and Technology Insertion*. NASA, 1998.

John Kelly, Ken Abernethy, Ann Sobel, James D. Kiper, and John Powell. Technology transfer issues for formal methods of software specification. *Software Engineering Education & Training, 2000. Proceedings. 13th Conference on*, 1:23–31, 1999.

John C. Kelly, Joseph S. Sherif, and Jonathan Hops. An analysis of defect densities found during software inspections. *J. Syst. Softw.*, 17:111–117, February 1992. ISSN 0164-1212. doi: 10.1016/0164-1212(92)90089-3. URL `http://dl.acm.org/citation.cfm?id=145095.145103`.

E. Kolb, O. Šerỳ, and R. Weiss. Applicability of the blast model checker: An industrial case study. *Perspectives of Systems Informatics*, 5947:218–229, 2010.

Kenneth Lausdahl. Translating vdm to alloy. In EinarBroch Johnsen and Luigia Petre, editors, *Integrated Formal Methods*, volume 7940 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38612-1. doi: 10.1007/978-3-642-38613-8_4. URL http://dx.doi.org/10.1007/978-3-642-38613-8_4.

Gerald Luttgen, Cesar Munoz, Ricky Butler, Ben Di Vito, and Paul Miner. Towards a customizable pvs. Technical report, NASA, 2000. PVS.

Petra Malik, Lindsay Groves, and Clare Lenihan. Translating z to alloy. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 377–390. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-11810-4. doi: 10.1007/978-3-642-11811-1_28. URL http://dx.doi.org/10.1007/978-3-642-11811-1_28.

PauloJ. Matos and João Marques-Silva. Model checking event-b by encoding into alloy. In Egon Börger, Michael Butler, JonathanP. Bowen, and Paul Boca, editors, *Abstract State Machines, B and Z*, volume 5238 of *Lecture Notes in Computer Science*, pages 346–346. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-87602-1. doi: 10.1007/978-3-540-87603-8_34. URL http://dx.doi.org/10.1007/978-3-540-87603-8_34.

Leonid Mikhailov and Michael Butler. An approach to combining b and alloy. In Didier Bert, JonathanP. Bowen, MartinC. Henson, and Ken Robinson, editors, *ZB 2002:Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 140–161. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43166-4. doi: 10.1007/3-540-45648-1_8. URL http://dx.doi.org/10.1007/3-540-45648-1_8.

Ponsard, Massonet, Molderez, Rifaut, van Lamsweerde, and Tran Van. Early verification and validation of mission critical systems. *Form Method Syst Des*, 30:233–247, 2007.

Ben Potter, David Till, and Jane Sinclair. *An Introduction to Formal Specification and Z*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1996. ISBN 0132422077.

Osamu Saotome, Edson Vinci, Moises Pereira Bastos, Sandro Shoiti Sato, and Lidia Hissae Shibuya. Itasat- desenvolvimento de um computador de bordo tolerante a falhas com comunicação padrão ccsds. Technical report, ITA, 2009.

Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Jr. Hunt, WarrenA. and StevenD. Johnson, editors, *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 127–144. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-41219-9. doi: 10.1007/3-540-40922-X_8. URL http://dx.doi.org/10.1007/3-540-40922-X_8.

Paulo Claudino Véras. Modelagem e análise do software embarcado de piloto automático de um vant. Master's thesis, ITA, 2007.

A.H. Watson, T.J. McCabe, and D.R. Wallace. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST special Publication*, 500(235):1–114, 1996.

J. Woodcock, P.G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR)*, 41(4):19, 2009.