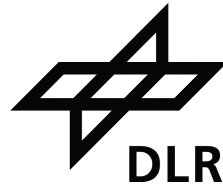

ROYAL INSTITUTE OF TECHNOLOGY
STOCKHOLM

MASTER DEGREE PROJECT

Design of an Automatic Specification-based Test-framework for On-board Software of Satellites

---

Author	Kilian Höflinger
Supervisor Company	Benjamin Weps
Company	German Aerospace Center
Examiner	Christian Schulte
Supervisor University	Gabriel S. Hjort Blindell
University	KTH – Software and Computer Systems
TRITA Number	TRITA-ICT-EX-2015:77

Abstract

Satelliter är sofistikerade och därför komplicerade konstruktioner som kräver tvärvetenskapligt lagarbete mellan olika experter från olika akademiska discipliner. Integrationen av specifika nyttolastkomponenter, liksom vetenskapliga experiment, med inbyggd programvara för satelliter är mycket utmanande. Domänexperten, som ägare av nyttolastkomponenten, besitter detaljerade insikter om hans eller hennes del, men saknar tillräckliga kunskaper i programmering för att implementera den i den inbyggda programvaran. Programmeraren är i stånd att skriva rätt kod för den inbyggda programvaran, men är oerfaren med nyttolastkomponenten. Denna rapport beskriver utformningen och genomförandet av ett automatisk, specifikationsbaserat testramverk för inbyggd programvara för satelliter för att överbrygga kunskaps- och kommunikationsklyftan mellan programmeraren och domänexperten. Modell- och testdriven utveckling är i fokus för testramverket. Med hjälp av ett domänspecifikt språk kan domänexperten modellera en specifikation i formell notation, som representerar potentiella användningsscenarier av komponenten. Dessa scenarier är automatiskt översatta till kompilerbara testfall i C++, som hjälper programmeraren att kontrollera den funktionella korrektheten av den inbyggda programvaran för nyttolastkomponenten när han eller hon programmerar den.

Satellites are sophisticated and therefore complicated constructs that require interdisciplinary teamwork of various experts of different academic disciplines. The integration of specific payload components, like scientific experiments, in the on-board software of the satellite is very challenging. The domain expert, as the owner of the payload component, possesses detailed insights on his or her component, but lacks sufficient programming skills to implement it in the on-board software. The programmer is able to write proper code for the on-board software, but is inexperienced with the payload component of the domain expert. This report describes the design and the implementation of an automatic specification-based test-framework for on-board software of satellites to bridge the knowledge and communication gap between the programmer and the domain expert. Model- and test-driven development are in the focus of the test-framework. With the help of a domain-specific language, the domain expert is able to model a specification in formal notation, representing potential use-case scenarios of the component. These scenarios are automatically translated to compilable C++ test cases, which help the programmer to verify the functional correctness of the on-board software implementation of the payload component while he or she is programming it.

Table of contents

List of figures ix

List of tables xi

Nomenclature xiv

1 Introduction 1

1.1 Motivation 1
1.2 Space Environment of the Test-framework 3
1.3 Objective 5
1.4 Strategy 7
1.5 Report Overview 9

2 Background 11

2.1 Packet Utilization Standard 11
2.1.1 CDH Structure 12
2.1.2 Packet Structure 13
2.1.3 Behaviour Determined by PUS 13
2.2 Definition of Testing 14
2.3 From Manual Testing to Keyword-driven Test-frameworks 14
2.4 Model-based Testing 16
2.4.1 Approaches in Model-Based Testing 16
2.4.2 Model-based Testing Taxonomy 17
2.5 Domain Specific Languages 24
2.5.1 Domain Specific Language Definition 24
2.5.2 DSL vs. GPL 24
2.5.3 Modelling with DSLs 24
2.6 The Development Tools 25
2.6.1 Eclipse Platform 25
2.6.2 Eclipse Modelling Framework 25
2.6.3 Xtext 27

2.6.4 Xtend 27
2.7 Temporal Logic 28
2.7.1 Kripke Structure 28
2.7.2 Linear Temporal Logic 29
2.7.3 Computation Tree Logic 31
2.8 Related Work 32

3 Project's Ethic Responsibility 35
3.1 Space Shuttle Challenger Disaster 35
3.2 Code of Ethics for Engineers 36
3.3 Ethical Aspects of the Project 37

4 Analysis and Requirements 41
4.1 General Requirements 41
4.2 Protocol Requirements 43
4.2.1 Package Structure 43
4.2.2 Behavioural Expressiveness Requirements 44
4.3 Additional Temporal Requirements 46

5 Design 47
5.1 Generic Top-Level Design Considerations 47
5.1.1 Test Dimensions 47
5.1.2 Test-framework Approach 49
5.1.3 Different Model-based Approaches 51
5.2 Application of the MBT Taxonomy 52
5.2.1 Model Class 52
5.2.2 Test Generation Class 53
5.2.3 Test Execution Class 55
5.2.4 Test Evaluation Class 55
5.3 Top-level Design 56
5.3.1 Data Description Language 57
5.3.2 Data Testing Language 57
5.3.3 Test Execution & Evaluation Platform 57
5.4 Data Description Language 58
5.4.1 Application-specific DDL 58
5.4.2 PUS-conform DDL 59
5.5 Data Testing Language 59
5.5.1 Data-centric Sub-grammar 60
5.5.2 Relational Expressions Sub-grammar 61

5.5.3 Modified Linear Temporal Logic Sub-grammar 62
5.5.4 Coverage Report 66
5.6 Test Execution & Evaluation Platform 66
5.6.1 Shared Components 66
5.6.2 OBSW Components 67
5.6.3 TEEP Components 68

6 Implementation 71
6.1 Used Tools & Implementation Workflow 71
6.2 Data-Description Language 71
6.2.1 DDL Grammar 72
6.2.2 Code Generator 74
6.3 Data-Testing Language 75
6.3.1 Data-centric Sub-grammar 75
6.3.2 Relational Expressions Sub-grammar 78
6.3.3 Modified Linear Temporal Logic Sub-grammar 80
6.3.4 DTL Structural Coverage Report 86
6.4 Test Execution & Evaluation Platform Implementation 88
6.4.1 Shared Components 88
6.4.2 OBSW Components 89
6.4.3 TEEP Components 90
6.4.4 Test Process 92

7 Evaluation 95
7.1 Automatic Specification-based Challenges Evaluation 95
7.2 Requirements Coverage Evaluation 96
7.2.1 General Requirements Coverage 96
7.2.2 PUS and Additional Temporal Requirements Coverage . 99
7.3 Limits of the Test-framework 103

8 Conclusion 105
8.1 Summary 105
8.2 Outlook 106

Bibliography 109

Appendix A Introduction 115

List of figures

1.1 The growth of the industry, displayed by the SFI between May 2013 and May 2015 . . . 2
1.2 Space mission phases . . . 4
1.3 Generic satellite hardware structure. . . . 5

2.1 PUS architecture . . . 12
2.2 Telecommand package structure . . . 13
2.3 Overview of MBT taxonomy of Zander, Schieferdecker and Mosterman . . . 18
2.4 Ecore components hierarchy. . . . 26
2.5 Mapping of Ecore to Jave elements. . . . 26
2.6 Overview of the workflow with Xtext. . . . 27
2.7 Kripke structure example: A man throws a ball vertically. . . . 29
2.8 Kripke model to explain expressiveness of CTL compared to LTL. 32

4.1 Telecommand packet structure . . . 44
4.2 Telemetry packet structure . . . 44
4.3 Packet structure of the TC/TM packet of the test-framework . . 44

5.1 Test dimensions . . . 48
5.2 Overview of the test-framework design and workflow. . . . 56
5.3 Overview of the Data Testing Language . . . 60
5.4 Behavioural part of the OBSW, represented as Kripke structure. 63
5.5 Overview of the Test Execution & Evaluation Platform . . . 67

6.1 Abstract syntax tree of DDL. . . . 72
6.2 System model example. . . . 73
6.3 Test model example. . . . 76
6.4 Abstract syntax tree of the Data-Centric sub-grammar. . . . 77
6.5 Abstract syntax tree of the relational expressions. . . . 79
6.6 Abstract syntax tree of the Modified Linear Temporal Logic. . . 80

6.7 Coverage report for the domain expert. 87
6.8 Reporter output of reportAll.xml. 91
6.9 Logger output of logAll.xml. 91
6.10 Flowchart of testing process. 93

7.1 Simple test model. 101

List of tables

2.1 Temporal and logical operators in LTL 30

4.1 General requirements of the test-framework 43
4.2 Expected behavioural expressiveness of the test-framework . . . 46
4.3 Additional temporal requirements of the test-framework 46

5.1 Temporal and logical operators in MLTL 64

6.1 Evaluation struct of the *TestEvalMeta* class. 84
6.2 Evaluation struct of the *TestEvalMeta* class. 86

A.1 Mission phases milestones 115
A.2 Satellite classification according to the mass 116

Nomenclature

AOCS	Attitude and Orbit Control System
ASBT	Automatic Specification-Based Testing
CDH	Command and Data Handling
CTL	Computation Tree Logic
DDL	Data Description Language
DLR	German Aerospace Center
DSL	Domain-Specific Language
DTL	Date Testing Language
ESA	European Space Agency
GPL	General-Purpose Language
LTL	Linear Temporal Logic
MBSD	Model-Based Software Development
MBT	Model-Based Testing
MLTL	Modified Linear Temporal Logic
NASA	National Aeronautics and Space Administration
OBSW	On-board Computer Software
SBT	Specification-Based Testing
SUT	System Under Test
TC	Telecommand
TEEP	Test Execution & Evaluation Platform

TM	Telemetry
TTC	Test Telecommand

Chapter 1

Introduction

This chapter introduces the topic and depicts the problem this report targets. It further outlines the main tasks, the goals as well as the purposes of this final degree project. The chapter closes with an explanation of the used scientific writing methods and a document overview.

1.1 Motivation

There is movement within space. Urgent space-related questions, like the future in human space-flight, are on the agenda of many governments. Depending on their answers, huge financial capital might be washed into the industry. The *Space Foundation Index (SFI)* displays the space industry in the U.S. public markets regarding its breadth and depth. Members of the SFI are on one side the manufactures of hardware, software or services, used to create space systems. And on the other, companies which exploit signals from space systems to offer a service to the customer, like TV broadcasting, remote sensing or mobile communication [1]. Figure 1.1 displays the development of the SFI between May 2013 until May 2015. Within the last two years the index grew by 40%, being able to keep up with high performance indices like the NASDAQ [1]. This steep growth reveals the current trend of this economical sector. The trend got further pushed by the market entry of famous entrepreneurs, like Elon Musk with his SpaceX falcon [3], who loudly claim market shares in the industry. Additionally, open innovation projects like CubeSat [2], which advertise their goals and challenges to a broad public, help to establish awareness for the industry. The strong market growth, the advertisements of open innovation projects and the unbroken public interest on Mars missions and human space-flight quicken this economic sector.

Fig. 1.1 The growth of the industry, displayed by the SFI between May 2013 and May 2015. Source: [1]

In 2013, the total amount of launched satellites rose by two thirds, in comparison to 2012 [4]. The reason for the increase cannot only be traced back to the aforementioned growth of the industry but also to a paradigm shift in the design of satellites. Nowadays, more than 50% of all launched satellites weigh less than 200 pounds. These micro-, nano- and pico-satellites often serve as a test platform for potential future technologies and have a rather short lifespan [4]. The strong economic growth of the industry and the paradigm change to smaller satellites challenge the market participants. On one side, they have to develop satellites faster and simultaneously maintain or even improve the quality of the space systems. On the other side, the shift from prototyping to industrial production has to be mastered wherever possible [5]. To overcome these challenges, standards, new approaches and innovative thinking is in need.

"***Software quality assurance is a planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements***" - ESA's software quality assurance definition [6] -

This report presents a design and an implementation of an automatic test-framework to ensure software quality for *on-board software (OBSW)* of satellites and additionally accelerate the development of the same. The framework bridges the communication gap between domain experts and programmers, and increases the software quality by a reduction of the threat of erroneous code. Therefore, this test-framework accepts the aforementioned challenges and follows the ESA's software quality assurance.

1.2 Space Environment of the Test-framework

The test-framework is used to verify OBSW of satellites. To better understand the application and the impact of the test-framework, it is useful to set it into context with the general life cycle of a space mission. Additionally, the generic hardware structure of a satellite is elaborated, as it is the target platform of the OBSW.

Space Mission Phases

The European Cooperation of Space Standardization (ECSS [7]) defined the space mission life cycle in their *Space project management* standard [8], dividing it in seven phases. Figure 1.2 reveals the phases graphically in a Gantt chart format, including the project milestones (a denotion of the milestones can be found in the appendix A). The phases are characterized form 0 to F [8]:

- **0:** Among other project top-level tasks, mission goals, preliminary technical requirements specifications and a preliminary risk assessment are elaborated.
- **A:** Basic concepts with the focus on feasibility are conducted. Mission goal crucial as well as critical tasks are identified and denoted in various project plans.
- **B:** A task division takes place among all project participants, including deadline definitions for the subcomponents. Therefore, interfaces are defined.
- **C:** The designs and concepts are finished and implementation as well as testing begins. In this phase the test-framework is used for the first time.
- **D:** The subcomponent is assembled and tested in integration and system tests. Here, the test-framework is also in use.
- **E:** Final checks take place before the launch and therefore the actual mission begins.
- **F:** The final phase is reached when the mission goals are accomplished or something unexpected and fatal happened. The only task is the disposal of the equipment.

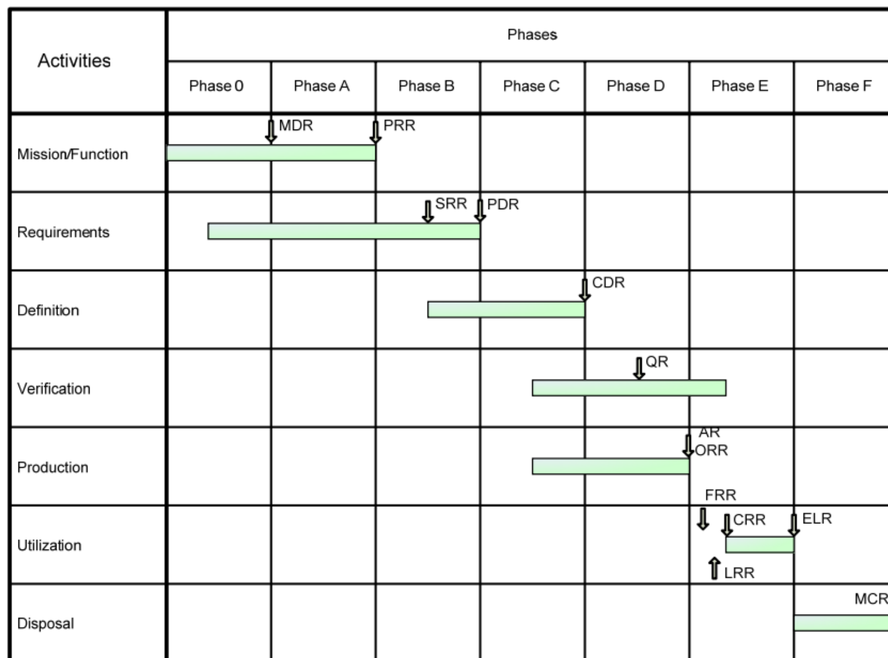

Fig. 1.2 Space mission phases. Source: [8, p: 19]

Generic Satellite Hardware Structure

The generic satellite hardware structure is elaborated by the usage of the BIRD (Bispectral Infra-Red Detection) mission, as an example. The BIRD mission has the goal to gain data on bush or forest fires on a global scale. Additionally, the bearing of volcanoes is evaluated. Infra-red sensors are used to measure the radiation at 4 and 10 micrometer wavelength, resolving temperature differences with a 0.5 K accuracy. The satellite got launched in 2001 and with its 92 kg it can be classified as a micro-satellite (Appendix A contains a list of the various satellite classes) [9, p: 667].

A satellite (space segment of a space mission) consists of two major components, the payload and the satellite bus. Figure 1.3 displays the generic hardware structure of a satellite graphically. The payload is the actual reason why the satellite is deployed in space. It depends on the mission goals and can comprise scientific instruments, transponders, meteorology instruments, navigation technology or military payload. On shared missions, different payloads can be combined on one satellite. Regarding the BIRD mission, the payload consists of a bi-spectral infra-red sensor system, an optoelectronic wide angle stereo sensor scanner, a data processing system and an artificial neuronal network classifier [9, p: 669].

The satellite bus is adapted to the payload. It takes care of generic tasks, which have to be executed on all satellites. The BIRD mission's satellite bus

consists of an *Attitude and Orbit Control System (AOCS)*, an on-board computer, a mechanical structure, an energy supply system, a telecommunication system and a thermal control system. Additionally, some satellites also have a propulsion system [9, p: 669].

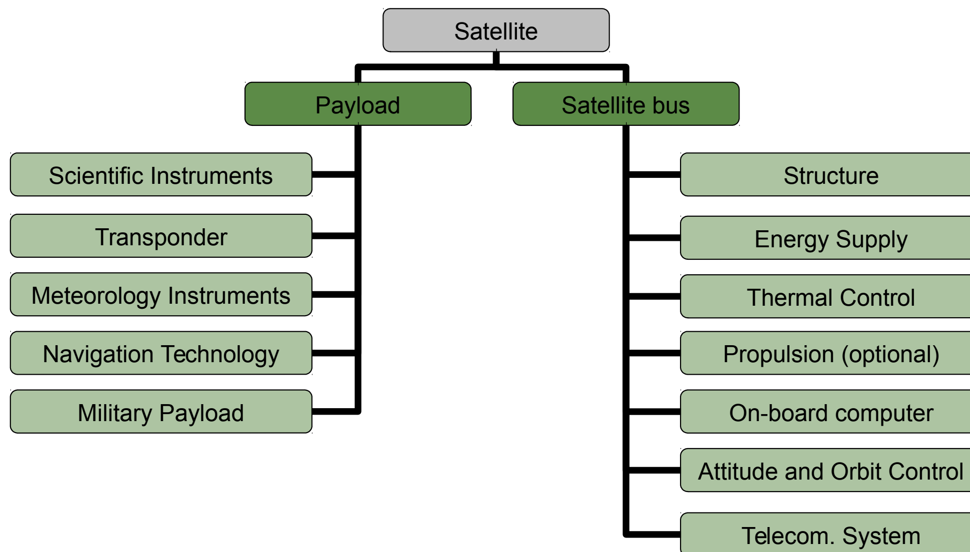

Fig. 1.3 Generic satellite hardware structure.

1.3 Objective

Problem Statement

Testing is an important aspect in the development of aerospace software. This is due to the difficulties for mission online software updates and the impact of programming errors, which – in the worst case – can lead to a total loss of the entire mission. The famous priority inversion bug of the Mars Pathfinder mission is an excellent example for that [10] [11]. Humans are likely to produce erroneous code. According to Steve McConnell, the software industry's average is about 15 – 50 errors per 1000 lines of delivered code [12]. Currently, OBSW tests are hand-crafted at the German Aerospace Agency (DLR).

Another problem is the poor re-usability of the OBSW, which leads to a drastic loss of knowledge already gained in previous missions. Institutes like the DLR and other space agencies are strongly project-orientated, with projects that run for years. It is not uncommon that employees leave the institute (change to another company, retire etc.), when their main project is finished. If that happens a drain of insights and know-how is mostly the result. The

National Aeronautics and Space Administration (NASA) tackled that problem with their Lessons Learned Information System, which got launched in 1994 [13]. Therefore, the preservation of know-how in vital concepts and the re-usage of knowledge is something that always needs to be improved, especially in a high-tech industry.

Also the growing complexity of the systems is problematic. Satellites are sophisticated constructs, which require interdisciplinary teamwork among experts of various domains (propulsion, thermal, data management etc.). Those experts define requirements for their components, which are a basis for the OBSW implementation of the programmer. This procedure is an iterative knowledge exchange process in which the domain expert provides requirements and the programmer returns specifications. It is iterative because the domain expert lacks sufficient knowledge on OBSW and the programmer lacks profound insights into the component of the domain expert, which can lead to miss understandings and therefore miss designs. The challenge is to bridge this knowledge gap by the creation of a, for both sides understandable, language to describe the requirements of the component of the domain expert.

Task

A concept for an automated test-framework on the basis of the *Data Description Language (DDL)* [14] has to be developed and implemented, which enables the execution of black box tests. Further, an analysis of a *telemetry (TM)* and *telecommand (TC)* transmission protocol, used in current space missions, has to be conducted. The result of this analysis is an additional definition of requirements for the test-framework.

The DDL is a DLR in-house-developed *Domain Specific Language (DSL)*. It is constructed with a data-centric approach. Data-centric means that the focus lies on the data and the data-flow in the system. Therefore, the DDL describes the data structures as well as the interfaces for the data transmission on the application and service level of the OBSW of satellites. Its purpose is to model the *Command and Data Handling (CDH)* subsystem of the satellite, which is in charge of the routing of TM and TC packets within the OBSW [14]. The concept of the test-framework should exploit the DDL and therefore get attached to it. To be able to test the functionality of the test-framework, chosen test cases have to be implemented. Concluding, a theoretical evaluation regarding the fulfilment of the requirements and the capability of the framework has to be conducted.

Purpose & Goal

The purpose is to bridge the communication gap between the domain experts and the programmers regarding the desired behaviour of the final satellite system. This gap exists due to a lack of a common language and knowledge basis. Test-driven development and automatic code generation has to be strengthened. And the code re-usability for follow-up missions ought to be increased.

The goal is to create a standard-compatible automatic test-framework. The framework has to be usable by the domain experts to define software specifications of their satellite components, which are automatically transformed into executable test-cases. These test-cases have to enable a test-driven development of OBSW. Therefore, the test-framework needs an execution platform which automatically executes and evaluates the tests against the OBSW under development.

Ethics and Sustainability

The ethical aspects as well as the sustainability of a project have a huge influence on its long term success. The report details the connection between theses aspects and this project in the Ethics Chapter (3).

1.4 Strategy

Method

The research for this project is conducted in a qualitative manner. An evaluation of various existing testing methods and frameworks is executed. Additionally, an analysis of a TC and TM protocol is conducted to derive data in the form of requirements for the test-framework. The basic philosophical assumption is the interpretivism, with an inductive research approach. The research is follows with the conceptual method [15].

Data Collection and Analysis

A qualitative data collection is carried out by the usage of academic search engines, like Google Scholar or the IEEE Explorer. Furthermore, an investigation of DLR in-house databases for scientific papers and articles is executed. The analysis of the potential sources is conducted using the three-pass approach.

In the first pass the quality and the relevance for the topic has to be evaluated. The second pass is focusing again on the quality and the relevance in a more detailed way, with an additional investigation of the references of the document. If the document overcomes also the second pass, a final evaluation takes place to incorporate the knowledge into the thesis [16].

Workflow

The workflow is subdivided into six phases. One day represents an 8 hours working day of one person.

1. **Orientation and organizational phase (21 d)**
   In this phase, the project planning, the acquirement of DLR specific knowledge and the initial literature research lies in the focus. Additionally, an initial presentation of the thesis topic is done.

2. **Interpretation and evaluation phase (19 d)**
   The interpretation and evaluation of the TC and TM protocol as well as the final literature research takes place. In the end, potential concepts should be prepared for a discussion with colleagues.

3. **Design phase (19 d)**
   This phase emphasises on the design of the concept, with an intermediate feedback round in the form of a midterm presentation and discussion.

4. **Implementation phase (21 d)**
   The implementation phase is focusing on the step-wise implementation of the concept.

5. **Evaluation Phase (10 d)**
   The evaluation phase is used to theoretically evaluate the automated test-framework regarding its requirements. Additionally, selected test applications are implemented and used as a practical demonstrator of the functionality and capability of the test-framework.

6. **Documentation phase (34 d)**
   In the final phase the results have to be documented. A major part of that is the establishment of the thesis, using the already written documents of the previous phases as a basis. Concluding, important documents like manuals and user-docs have to be written.

1.5 Report Overview

The Introduction Chapter displays the motivation for this project and the targeted problems are elaborated. The task is presented and the purpose as well as the goal of the project denoted. Additionally, the strategy, based on the scientific method, the data collection and the workflow within this project, is documented. The subsequent Background Chapter provides the reader with sufficient project specific knowledge and states the related work in the domain. Following, the Analysis and Requirements Chapter reveals the requirements for the project. These requirements are applied in the Design Chapter to define the layout of the test-framework, under consideration of various options. Two emphasises are within this Chapter, one on the derivation of the top-level design and the other on the theoretical structure of the components of the test-framework. The output of this Chapter is used in the subsequent Implementation Chapter to show how the theoretical design got transformed in an executable test-framework. The evaluation of the implementation is based on knowledge of the Background Chapter and the requirements of the Analysis and Requirements Chapter as well as on the problem statement of the Introduction Chapter. In addition, identified limitations of the test-framework are explained. Lastly, a summary of the project and a future outlook is provided in the Conclusion Chapter.

Chapter 2

Background

The primary goal of the Background Chapter is to provide information which serves as a basis for the upcoming chapters. The reader gets informed about procedures, standards, hardware structures and software structures in the field of the space industry, which are in context to this report. Afterwards, basic testing knowledge is provided, focusing on automated frameworks and model-based testing in particular. Domain Specific Languages and modelling with them is elaborated and the Eclipse Modelling Framework (EMF) with the Xtext framework presented. Finally, temporal logic with a focus on Kripke structures, Linear Temporal Logic and Computation Tree Logic is introduced. The related work in the domain of this test-framework closes this chapter.

2.1 Packet Utilization Standard

The Packet Utilization Standard (PUS) [17] is a customizable standard, defined by the European Cooperation for Space Standardization (ECSS, http://www.ecss.nl/). It can be seen as a guideline to develop OBSW of satellites with a focus on the communication. The term OBSW refers in the context of PUS and the entire subsequent report to the OBSW in the user-mode. To apply the standard it has to be tailored to the specific space mission requirements. The necessity of tailoring lies in the vast range of requirements of space missions, especially scientific ones. PUS describes three main aspects: the structure of the OBSW, the communication package structure and the behaviour of the OBSW [17].

2.1.1 CDH Structure

PUS defines the structure of the software, which is mirrored in the CDH subsystem of the OBSW. Figure 2.1 displays the structure graphically. The CDH subsystem can be seen as the heart of the user mode OBSW. It is subdivided in three layers. On the top layer are the application processes, the middle layer is occupied by the services and the base layer consists of the sub-services (packets). The sub-services contain the payload parameters. This structure determines the routing of the packets from the application-level down to the sub-service-level.

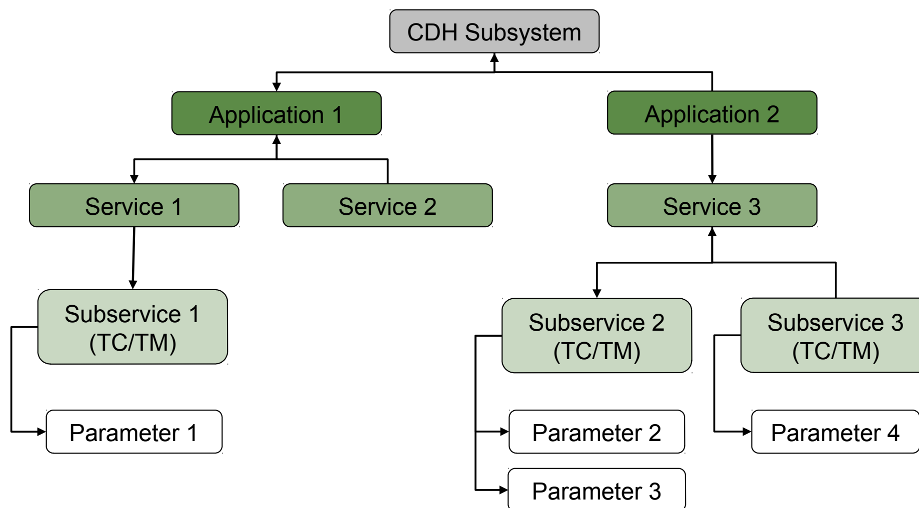

Fig. 2.1 PUS architecture

The application processes control the hardware systems of the satellite and the specific features of them, invoking system actions by the usage of hardware drivers. Additionally, they hold interface references to the services. An application process can be contacted via its unique *application process ID (APID)* [17].

The services host the sub-services and can be reached over the application via references. Services can be used by one or more applications. Identification takes place by a *service ID (SERID)*. There are already services defined in PUS, which are very common on spacecrafts, like the *Housekeeping Service* or the *Memory Management Service*. Additional information on how to design custom services is provided [17].

The sub-services are either TC or TM packets. These data structures are parameters for methods which are directly integrated in the services. TC packets are transmitted from the ground station to the satellite and TM packets

vice versa. Identification takes place via the *sub-services ID (SUBSERID)*. When a sub-service is triggered by the arrival of a TC packet, it modifies the state of the application process. The application then reacts on this modification accordingly [17].

2.1.2 Packet Structure

The second major part that is standardized is the structure of the TC & TM packets. Both packet types consist of a *Packet Header* and a *Packet Data Field*. Depending on the requirements of the mission, some header information may get used, some may not (remember PUS has to be tailored before usage). The maximum packet size is 65KB, due to limited connection time, limited link bandwidth and recovery time from failure. In case of a larger payload, the data can be grouped or sequenced, using the specific parameter of the *Packet Header*. Figure 2.2 displays the structure of a TC packet. In the Analysis and Requirements chapter, the TC and TM packet structures are analysed to determine requirements for the test-framework [17].

Packet Header (48 Bits)							Packet Data Field (variable)			
Packet ID				Packet Sequence Control		Packet Length	Data Field Header (Optional)	Application Data	Spare	Packet Error Control
Version Number (=0)	Type (=1)	Data Field Header Flag	Application Process ID	Sequence Flags	Sequence Count					
3	1	1	11	2	14					
16				16		16	Variable	Variable	Variable	16

Fig. 2.2 Telecommand package structure

2.1.3 Behaviour Determined by PUS

The PUS describes the behaviour of the OBSW to a certain extent. Options and modes of the OBSW are defined [17]. A reference to the Analysis and Requirements Chapter shall be given, in which the implemented behavioural aspects of the PUS in the test-framework are elaborated in detail.

2.2 Definition of Testing

M. Uttinger and B. Legeard give a definition in their book *Practical Model-Based Testing - A Tools Approach* [18] which is used for the rest of the report.

"*Testing is a **dynamic** verification of the behaviour of a program on a **finite** set of test cases, which get **selected** from the in general infinite cases domain, against the **expected** behaviour.*" - Definition of Testing by Uttinger and Legeard [18, p: 3] -

Dynamic means that we execute the program with the input predefined by the various test cases. In contrast to that, static testing of a program is without its execution (code reviews, walkthroughs, etc.). The **finite** set of test cases reveals the incompleteness of testing. Problematic for testing are programs with side effects as they mostly result in a state space explosion (reactive systems). This can lead to an infinite amount of test cases, which lacks the feasibility of testing it completely [19, p: 11]. Hence the **selection** of a sufficient set of test cases is one of the key challenges, which is evaluated by a predefined coverage criteria. Strategies like finding equivalence classes and boundary value testing are just two of many in use. The final step is to resolve the oracle problem, which is the determination whether the test outcome is a success or a failure by comparing it to the **expected** outcome (oracle) [18, p: 3-4].

2.3 From Manual Testing to Keyword-driven Test-frameworks

Manual testing is a process in which the tester searches directly and manually for defects in the *System Under Test (SUT)* without the usage of any automation supporting concepts or tools [20]. An automated test-framework is a collection of assumptions, concepts, processes and environments used to design, execute and evaluate software tests in an automatic fashion [21]. Testing evolved over time, offering the possibility to subdivide it into four main generations: Manual testing, linear script-based testing , data-driven testing and keyword-driven testing.

Manual and Capture & Replay Testing

In manual testing the entire testing process is hand-crafted. The test case design often follows an informal test plan, which has to be translated in input for the SUT. The execution is conducted by the usage of that input directly on the SUT. Also the result of the test case is evaluated manually as well as the fulfilment of the test coverage requirements. Manual testing requires a deep understanding of the SUT and its expected behaviour. In case of a huge and complicated SUT, the testing process becomes very time- and money-intense [18, p: 21].

To reduce the costs and the required time for the test execution, the capture & replay approach has been developed. In this approach, the test case execution gets recorded once, to be able to replay the test case execution on later versions of the SUT. The test cases itself are still hand-crafted. The outputs of the test runs on the various versions can be logged and compared to each other. This first semi-automatized approach is often used for GUI tests. Problematic is the sensitivity of the captured tests towards API changes of the SUT, making it almost impossible to maintain multiple versions, especially when the comparison of logged output is used to evaluate the correctness of the current version [22, p: 62].

Script-based Testing

Script-based testing focuses on the execution and evaluation of the test cases. The script itself is executable and runs one or more test cases. It interacts directly with the SUT. The conducted tasks of the script are: initialization by a state change of the SUT, execution of the tests directly on the SUT and comparison of the result to derive a verdict on the success or failure of the test [22, p: 65].

The script-based testing automatizes the execution and evaluation of the testing process. But the direct interaction with the SUT makes it almost as difficult to maintain as the capture & replay approach. The input data resides within the scripts as well as at least the API of the SUT, sometimes also complete source code sections of the SUT. The poor re-usability of the scripts leads to a fast rank growth of scripts, which are hard to read and to maintain [18, p:24].

Data-driven Testing

In the data-driven approach, the input data of the script is stored in separate files. This enables the usage of the same script for multiple input data, which

is reducing the overall amount of scripts. But the general poor re-usability of linear scripts is not tackled by a data-driven approach. Nevertheless, the overview is better, due to a clear separation of code and data [23].

Keyword-driven Testing

The keyword-driven (also called: table-driven, action-word-driven) approach introduces an abstraction layer between the tests and the SUT. The test scripts contain action words which are associated to functions of the SUT or scripts that can be executed on the SUT. The relation between the action words and the scripts/functions is determined in a table, which serves as an adapter between the test script and the SUT. To run a test script, a test execution tool is necessary [18, p:26].

The keyword-driven approach offers more independence of the SUT. The action word layout of the test scripts allow also non-programmers to read and write the tests. An additional advantage is that a change in the SUT can often be handled by a change in the adapter. Nevertheless, changes have to be implemented manually as well as the verification of the test-coverage [18, p:28].

2.4 Model-based Testing

Model-Based Testing (MBT) belongs to the family of keyword-driven testing, as a model is nothing but an abstraction of the underlying system. There are four major approaches in MBT.

2.4.1 Approaches in Model-Based Testing

1. **Generation of test data from a domain model:** The input is derived from the domain model. The goal is to find an efficient subset of all possible inputs to meet the coverage requirements of the test. The oracle problem remains unsolved [18, p: 6].

2. **Generation of test cases from an environment model:** The subset of all possible inputs is derived from an environment model of the SUT. Also here, the oracle problem remains unsolved [18, p: 6].

3. **Generation of test cases with oracles from a behaviour model:** In this approach the input as well as the expected output are created from a behaviour model of the SUT. To be able to incorporate this oracle

information in the model, the test designer requires knowledge about the relationship between the input and the output [18, p: 6].

4. **Generation of test scripts from abstract tests:** Abstract test cases are given, which are transformed in executable test scripts. The information on the structure and the API of the SUT for the executable test script is gained by the system model [18, p: 6].

2.4.2 Model-based Testing Taxonomy

A taxonomy classifies a topic in ordered categories. This is very helpful to keep the oversight on huge software constructs, like an automatic test-framework. Taxonomies display options and reveal dependencies based on the selected options. They help to develop a correct design from the beginning and therefore prevent deadlocks in the implementation phase.

The basic *Model-Based Testing (MBT)* taxonomy was introduced by Utting, Pretschner and Legeard in 2006, in their working paper "*A taxonomy of model-based testing*" [47]. At this stage, the taxonomy does not have a specific addressee. It can be seen as a generic concept to derive characteristics for an implementation of a MBT framework.

In 2009, Zander-Nowicka used the taxonomy as a basis for her doctoral dissertation. She enriched the existing taxonomy to fulfil the specific needs of real-time embedded systems in the automotive industry [48].

This was taken by J. Zander, I. Schieferdecker and P. J. Mosterman in their book "*Model-Based Testing for Embedded Systems*" *(2012)* as footing [46, p: 3]. They changed the taxonomy of Zander-Nowicka to make it usable for embedded systems of multiple industry domains. Figure 2.3 displays the overview of their taxonomy.

Figure 2.3 reveals the structure of the MBT graphically, with the classes as the root elements and specific categories as their childes. On the lowest level are the options, which represent selectable technologies/approaches/algorithms of the categories. The mentioned exemplary options do not comprise all available options, but represent a list of commonly used ones within the field of MBT. It is possible to select multiple options within one category, in case they are applicable together [46, p:8].

1. The Model Class

The Model class contains the MBT basis, as well as the specific model properties as categories.

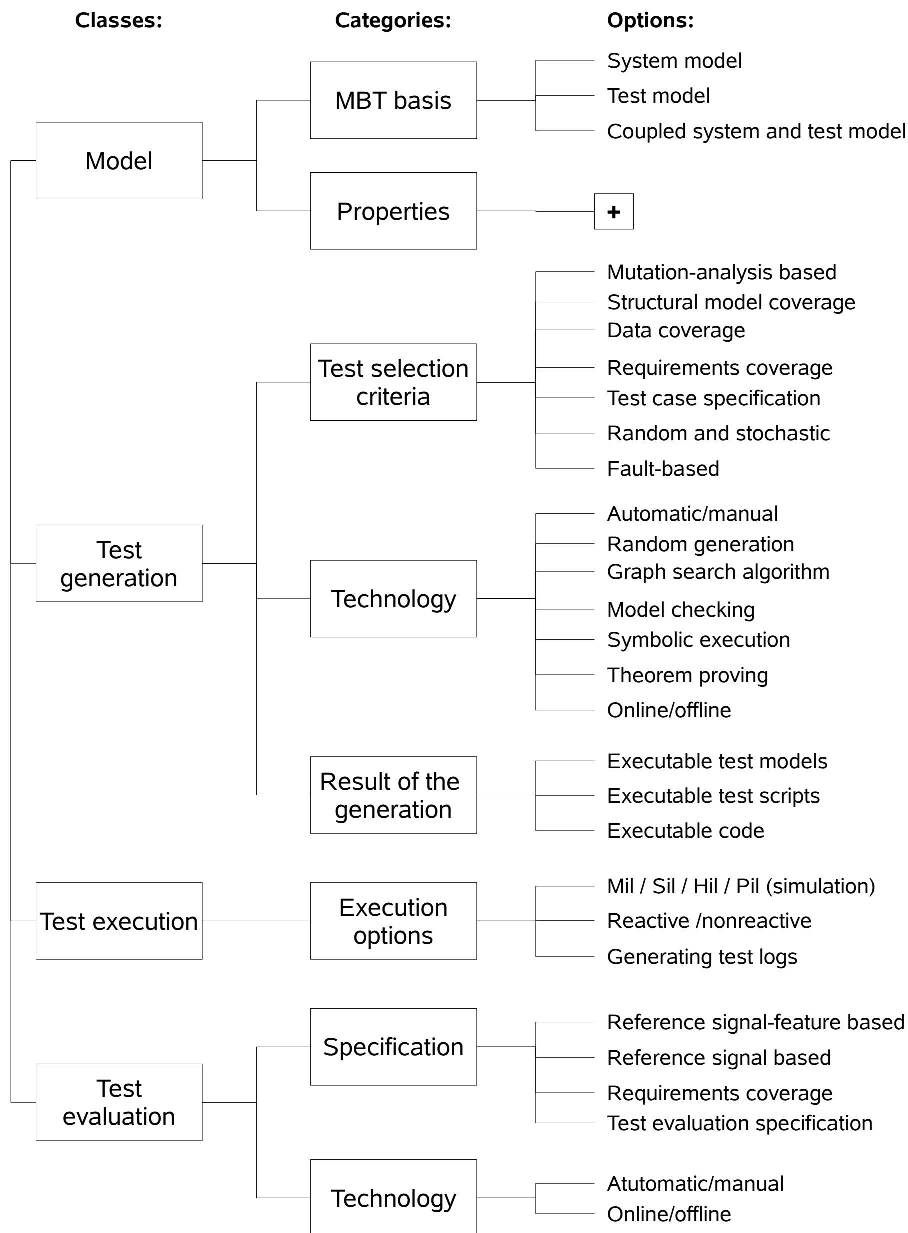

Fig. 2.3 Overview of MBT taxonomy of Zander, Schieferdecker and Mosterman. Source: [46, p:8]

MBT Basis

To be able to derive tests, different sources to create test cases can be used. It is possible to create tests using the information obtained within the *system model*, which is the abstract representation of the SUT. Another option is to extract the test cases from a specific *test model*, which only contains test relevant data. This can be also an environment model, but it is not considered explicitly by the authors. The third option is to use a *coupled system and test model*, linking the *system model* with the *test model* [46, p:8].

Properties

The properties describe the abstraction level of the model and cluster requirements for it. In case of a specific domain model, the domain requirements for the model are listed here. These requirements are mostly soft requirements which offer a certain interpretation freedom.

2. The Test Generation Class

The test generation class focuses on all aspects which are related to the creation of the test cases. It is subdivided into three categories: test selection criteria, technology and result of the generation [46, p:9].

Test Selection Criteria

The test selection criteria determine how the tests are selected from the source, which is in our case the *coupled system and test model*. And how to evaluate the completeness of the generated test suite regarding a specified coverage criteria. Those two criteria have to fit each other and get combined to one category. There are several options within this category, following different approaches. Underneath, frequently used ones are listed [46, p:10].

- **Mutation-analysis based:** There exist different versions of this option, but they all share a mutation of the system model or the test model. In case of a system mutant, which is a slightly modified version of the original system model, the output of the test on the mutant is compared to the output of the not mutated one. In case they match, the mutant is marked as equivalent. Another alternative is to modify the test model. There, the input data is altered to determine ranges and borders of the system model [46, p:10].

- **Structural model coverage:** Within this approach, the control-flow of the system model is investigated to determine the structure and to generate test cases out of this information. A pre-defined coverage criteria, regarding the control-flow, defines the termination of the test generation [46, p:10].

- **Data coverage:** The central part of this approach is a data range decomposition to reveal equivalent classes and a boundary value analysis on those classes. Testing takes place on the boundaries as well as on

one representation of every identified equivalent class. The coverage is determined by the range of data, which is analysed [46, p:10].

- **Requirements coverage:** This simply sets an informal requirements coverage as the coverage criteria as well as the test selection criteria [46, p:10].
- **Test case specification:** The test specification, which is created by the test designer, is used to derive and instantiate test cases of it. The coverage is manually determined by the test designer and therefore the test coverage criteria [46, p:10].
- **Random and stochastic:** This approach is mostly applied to environment models. There is a huge variety of algorithms which are used to create the input data, in case of random as well as stochastic. Determining a meaningful coverage criteria is difficult, especially when a random approach is used. Mostly a specific amount of test cases stops the testing [46, p:10].
- **Fault-based:** This approach is based completely on the insights and experience of the test designer, who tries to cover the errors that might occur. The coverage criteria is basically the knowledge of the test designer about the potential failure sources [46, p:10].

Test Generation Technology

The subsequent mentioned technologies can be applied to create the test suite. They are highly depending on the prior decisions in the test selection criteria. This is due to the fact that not every technology can be combined with all test selection criteria. But it is possible and even recommended to combine test generation technologies to exploit automatism and other wanted effects [46, p:12].

- **Automatic/manual:** Automatic test generation is based on the automatic creation of executable tests by the usage of test specifications. Manual test generation is the same, but the creation is done by hand and not automatic [46, p:12].
- **Random generation:** The random generation technology refers directly to its counter part in the test selection criteria category. It is simple to implement but not predictable in case of execution time to reach the specified coverage [46, p:12].

- **Graph search algorithm:** This covers a collection of algorithms to explore the nodes or/and the transitions of a graph, representing the system model. Famous example algorithms are the Chinese postman or Depth-first-search. Many coverage criteria can be applied: all nodes, all transitions, all cycles etc. [46, p:12].

- **Model checking:** The goal is to verify or falsify certain properties of a model. The focus lies in general on the verification that something bad never happens. In case a property is falsified, the model checker returns a counterexample. This counterexample is the test case [46, p:12].

- **Theorem Proving:** Theorem Provers and Model checkers are closely related to each other and sometimes replaceable by each other. But theorem provers generally check formulas on satisfiability which are included within the system model [46, p:12].

- **Online/offline:** Online generation technologies can react to state changes of the SUT while runtime. That means test cases can be created while the SUT is under execution, to execute them in the same test run. Offline generation technologies are not able to do that while runtime. The creation of the entire test suite has to be completed before the execution of the test begins [46, p:12].

Result of the Generation

There are three potential results of the test case generation: Executable test models, executable test scripts and executable code [46, p:12].

3. The Test Execution Class

The test execution class has only one child category: the execution options. This category determines the development stage of the system on which the tests "loop". Additionally, reactive and non-reactive execution is considered and the generation of logs [46, p:13].

Execution Options

- **Model-in-the-Loop (MiL):** Required is a behavioural model on which the tests are executed, to enable a development process early evaluation of the functionality of the design [46, p:13].

- **Software-in-the-Loop (SiL):** The software implementation of the entire system is the SUT. In our case the software implementation is the OBSW and the entire system is the satellite [46, p:13].

- **Processor-in-the-Loop (PiL):** One level above the SiL is the PiL. There the software is tested on the hardware. In case of an embedded system, the software is tested, using the micro-controller it is designed for [46, p:13].

- **Hardware-in-the-Loop (HiL):** The SUT is the complete hardware-software system with the peripherals under test and a simulated environment [46, p:14].

- **Vehicle:** The final development stage for an SUT is the entire final hardware and software system, tested in the actual environment [46, p:14].

- **Reactive/Non-reactive execution:** In case a reactive execution is desired, the execution platform has to react on the output of the SUT and decides which test case should be executed next. This comes often along with an online generation of test cases [46, p:14].

- **Generating test logs:** Test logs can be produced for each run and reused to evaluate the coverage. They contain detailed information on the steps and the order of the execution of the test cases on the SUT [46, p:15].

4. The Test Evaluation Class

Within this class the results of the test cases are evaluated. The goal is to compare the output of the test cases to the specific oracle, to derive a verdict. This class is subdivided into two categories, the specification and the used technology [46, p:14].

Specification

The specification of the test evaluation is needed to determine the source of the oracles and which characteristic of the output of the test cases are used to derive the verdict [46, p:14].

- **Reference signal-based specification:** The output signal value of the SUT is compared directly to an oracle signal value to assess the behaviour of the SUT [46, p:14].

- **Reference signal-feature-based specification:** To do a signal-feature-based specification, it is necessary to determine the features of the output's signals. Signal features are attributes of a signal, for example a maximum or an increase [46, p:15].

- **Requirements coverage criteria:** This criteria is used to evaluate the coverage of informal requirements regarding the actual behaviour of the SUT. The requirements should be traceable towards the test model and the SUT, to simplify the evaluation [46, p:16].

- **Test evaluation definition:** The specification-based output (oracle) is evaluated against the actual output of the SUT. The test designer is able to determine which oracle has to be evaluated against the output at a specific point in time. Formal notations for the test case specification are often used [46, p:16].

Technology

The technology used within the test evaluation class has to be selected whether the test evaluation should be a manual or an automatic process and if the evaluation should take place in an offline or online manner [46, p:16].

- **Automatic/Manual technology:** As already mentioned in the test generation class the distinction between the automatic and the manual technology is how a predicate is created. In the test evaluation class, the focus lies on the oracle, if it got created manually by the test designer or automatically by a deduction of the behavioural model. In case of an automatic creation of the oracles, this step is in general already executed within the test generation. Another point is the manner how the result is compared to the oracle and therefore assessed [46, p:16].

- **Online/Offline execution of the test evaluation:** This option is used to distinguish the point in time when the output of the result is assessed. In case of an online evaluation, the verdict is created when the SUT is still under execution. On the other side, in case of an offline evaluation the collected outputs of the test cases are assessed after the execution of the SUT, when all the test cases are executed [46, p:16].

2.5 Domain Specific Languages

2.5.1 Domain Specific Language Definition

A domain is a class of problems. A *Domain-Specific Language (DSL)* is a language which can be used to describe these problems. A *General-Purpose Language (GPL)* covers bigger or multiple domains. The DSL's terminology is close to the one in the domain to enable a precise representation of the abstraction used in the domain. Most DSLs are text-based, but also graphic and symbolic elements get applied [24, p: 28].

2.5.2 DSL vs. GPL

A clear categorization of a language whether it is a DSL or a GPL is often not easy. Some languages may appear in one direction as a DSL and in the other as a GPL. The following characteristics are therefore soft characteristics which have to be evaluated in every specific language case to derive a conclusion.

GPLs cover larger and more complex domains and have a larger language size. They are almost always Turing-complete. Their lifespan can range from years to decades and they evolve slowly, often standardized. The language developer can be a guru or a committee and the user community is large and anonymous.

DSLs focus on smaller and well-defined domains, what mirrors in the language size and layout. The evolution velocity is high, driven by only a few engineers and domain experts. DSLs are mostly not Turing-complete. The lifespan is generally shorter as for GPLs and the user community is smaller and local [24, p: 31].

2.5.3 Modelling with DSLs

DSLs are often used for modelling purposes. To enable *Model-Based Software Development (MBSD)* a prescriptive model is needed, which can be used to generate executable code. Most important is to translate the DSL and the defined model instance in a meta model, which can be used to transform it into the executable code. To enable this in a facile way, tool support is needed. Such a modelling approach with tool support allows an easy definition of the notation of the model. The code generator/compiler is customizable and the *Abstract Syntax Tree (AST)* easy accessible to modify it on the fly. Sophisticated modelling frameworks offer various additional features that facilitate the development of

DSLs for modelling [24, p: 33]. The Xtext modelling framework is such a tool. It is used for the implementation and presented in the subsequent section.

2.6 The Development Tools

2.6.1 Eclipse Platform

The Eclipse Platform contains the functionality to design and build *Integrated Development Environments (IDEs)* and other tools. One of the core advantages of IDEs and tools, developed on this platform, is the possibility to integrate them with existing platform conform projects. The Eclipse Platform contains the *Rich Client Platform (RCP)* as a subset, which clusters projects that do not target classic software development. Members of the RCP act for example in the field of the automotive industry, medical applications or space exploration [25].

The most famous Eclipse Platform-based IDE project is the *Software Development Kit (SDK)*. Its development is mainly influenced by the Eclipse Platform itself, the *Java Development Tools (JDT)* project and the *Plug-in Development Environment (PDE)* project [25].

2.6.2 Eclipse Modelling Framework

The *Eclipse Modelling Framework (EMF)* targets the development of model-based software. The goal of EMF is to generate software code for the implementation from a user-defined structured data model. The structured data model is instantiated in a custom-developed editor, which is based on a user-defined language [26]. The EMF model can be exploited to directly generate code from it, using tools like Acceleo, Jet or Xtend [24, p: 274].

The core component of the EMF on the modelling side is the Ecore meta-meta-model. Ecore is basically a subset of UML Class diagrams, and is therefore mappable to many *Object-Oriented (OO)* programming languages [26]. The goal of the design of Ecore was to facilitate the translation between the in the Ecore format defined model in executable source code [27]. Figure 2.4 displays the hierarchy of the Ecore language (green components are abstract). And Figure 2.5 denotes the mapping to the Java language, as an OO programming language example.

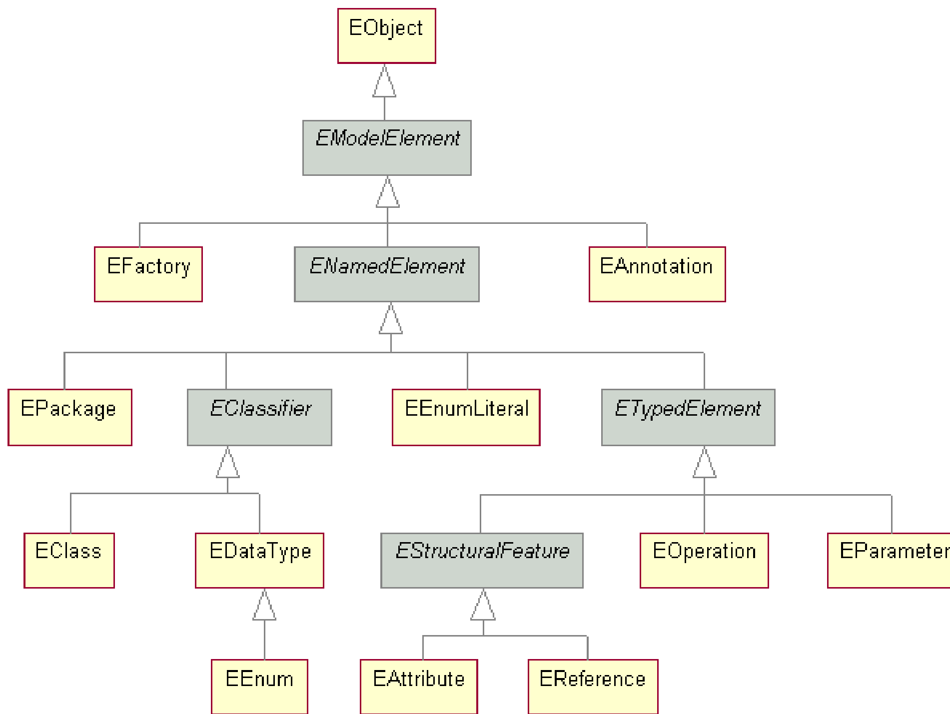

Fig. 2.4 Ecore components hierarchy. Source: [28]

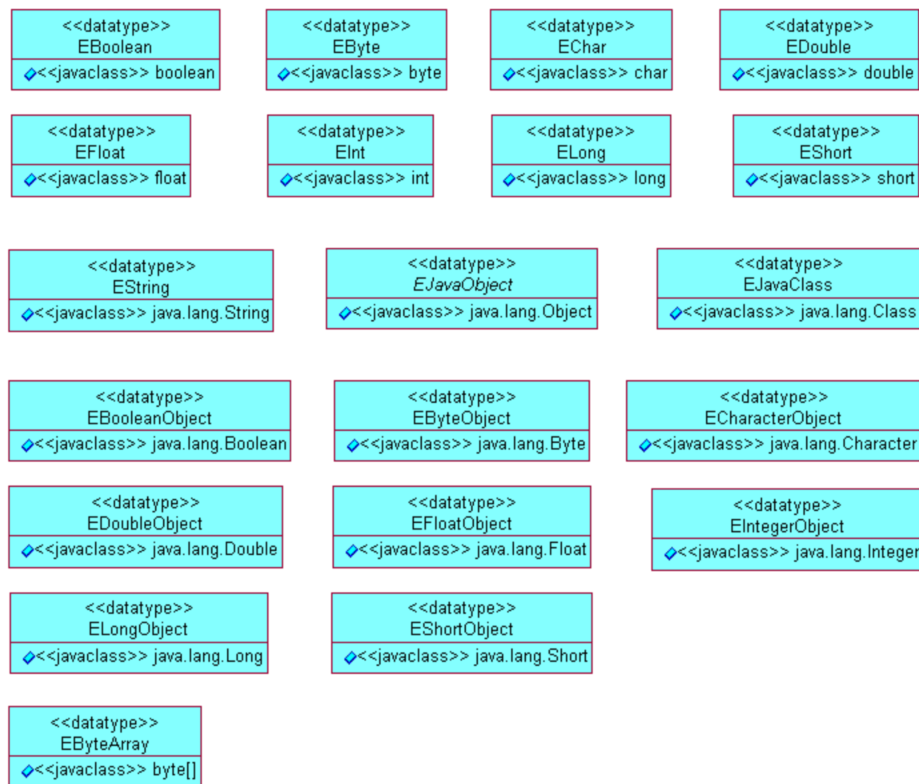

Fig. 2.5 Mapping of Ecore to Jave elements. Source: [28]

2.6.3 Xtext

Xtext is an open-source framework to develop GPLs as well as DSLs and is fully integrated in the Eclipse Platform. With the Xtext framework, grammars can be described in *EBNF (Extended Backus-Nauer Form)* style. It derives the AST from the user-defined grammar as an instance of the Ecore language by an integrated parser [24, p: 196]. It therefore builds up on the EMF. Different grammars can be cross-referenced to be able to use instantiated objects of one model in another. To instantiate the model, an editor – which runs as an SDK plug-in – is generated from the grammar. Exemplary features of the editor are: syntax highlighting, code completion, code folding and static syntax validation. The framework also offers the option to easily include semantic validation, explicit scoping and code generation [29]. Semantic validation is used to verify the model on its semantics while the user generates it, based on constraints which have to hold [24, p: 239]. The explicit scoping is important for the correct referencing of already defined objects of the model or of a cross-referenced model [24, p: 230]. The code generation is conducted using Xtend, which accesses the AST and translates the model instance into the desired format [24, p: 274]. Figure 2.6 shows an overview of the workflow with the Xtext framework.

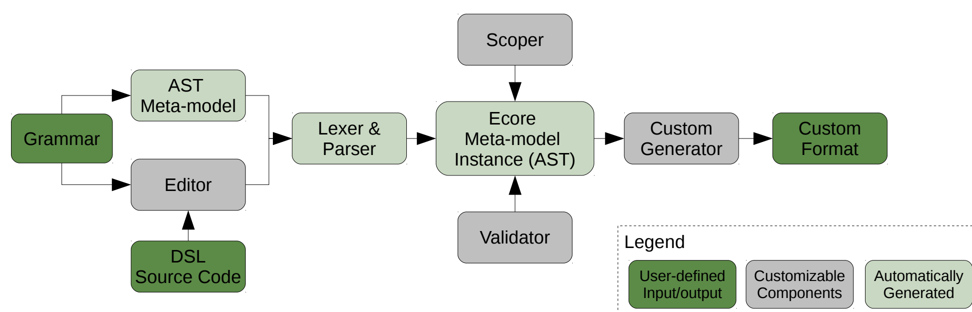

Fig. 2.6 Overview of the workflow with Xtext.

2.6.4 Xtend

Xtend is a statically-typed, general purpose, high level programming language and a dialect of Java. It is possible to mix Xtend and Java within one project without interoperability problems. The code is translated directly in readable Java code and not in Java byte code. It has some specific features like operator overloading, type inference or powerful switch expressions, that Java currently does not have [30]. To automatically transform the Xtend source code in Java source code, a compiler is provided and additionally an interpreter.

Xtend itself is built with the Xtext framework and therefore offers the powerful Xtext-based editor.

It can be used for code generation and model transformation. In the case of the Xtext framework, it's main fields of application are the validator, the scoper, the formatter and the code generator. The developers of Xtend emphasized on the template expressions, to enable a faster and easier code generator development [24, p: 274] [30].

2.7 Temporal Logic

Temporal logic is a formal notation to describe systems like programs. In contrast to informal notations, the ability to express the system in temporal logic is rather small. But the possible statements about the system are completely unambiguous. Temporal logic has it's roots in the propositional calculus and adds temporal operators to it [31].

In software development it is common not only to divide the entire development process in smaller processes (e.g. Waterfall model) but also to divide the software in hierarchical abstraction layers. An UML model is an abstraction of the source code. Arbitrary many abstraction layers can be integrated between the first top-level design and the final source code implementation.

Temporal logic can be applied on any of these hierarchical abstraction layers, whereas source code is also only an abstraction layer. A statement, described in temporal logic, about the program is a meaningful statement on all lower layers of the program/system [31].

2.7.1 Kripke Structure

A Kripke structure can be used to describe reactive systems. A reactive system changes its internal state, based on external stimuli for the system, like OBSW of satellites. Also the description of a model of the system is possible, as it is part of the temporal logic and therefore applicable on any hierarchical level of the entire system.

A Kripke structure is a graph with reachable states as the nodes and transitions as the edges. Deadlock states, with no escape-transition, are not allowed and are resolved by a transition to the deadlocked state itself. Additionally, the states have labelling functions, which are sets of *Atomic Propositions (APs)* that have to be true in that state. An AP can only evaluate to true or false and it is

user-defined [32, p:45].

A Kripke structure is a tupel $\kappa = \langle S, S_0, R, L \rangle$ for a finite set of AP with:

- S : Set of states;
- S_0 : Set of initial states with $S_0 \subseteq S$;
- R : Transition relation with $R \subseteq S \times S$
- L : Labelling function with $S \rightarrow 2^{AP}$

To illustrate a Kripke structure, an example is provided (Figure 2.7): A man throws a ball vertically in the sky and catches it again (under the assumption that the ball never reaches the escape velocity). The APs are the kinetic and potential energy levels in the ball and attached via the labelling functions to the state.

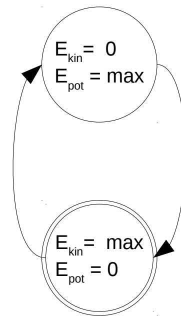

Fig. 2.7 Kripke structure example: A man throws a ball vertically.

2.7.2 Linear Temporal Logic

Linear Temporal Logic (LTL) is used to describe properties that have to hold on traces of a system and is widely used in model verification. The primary version consists of a set of atomic propositions, the logical operators (boolean connectives) $\wedge$ and $\neg$ and the temporal operators (modal operators) "next"($\bigcirc$), "always"($\Box$), "eventually"($\Diamond$) and "until"(U). The temporal operators determine the point in time and therefore the state in which the atomic propositions have to hold. The logic operators connect and logically manipulate LTL formula [33].

Syntax

To understand LTL, the connection between atomic proposition, formula and operator has to be stated clear:

- If the atomic proposition p is an element of the set of atomic propositions APS, then p is a formula ϕ;
- If ϕ and ψ are formulae, then $\phi \wedge \psi$, $\neg\phi$, $\bigcirc\phi$, $\Box\phi$, $\Diamond\phi$ and $\phi U \psi$ are also formulae;

Over the time new operators got introduced to the calculus. Table 2.1 explains common temporal as well as logical operators.

Operator Type	Syntax	Explanation
Temporal	$\bigcirc$ ("next") ϕ	ϕ has to hold at the next state;
Temporal	$\Box$ ("always") ϕ	ϕ has to hold in all future states;
Temporal	$\Diamond$ ("eventually") ϕ	ϕ holds eventually in a future state;
Temporal	ϕ U ("until") ψ	ϕ has to hold at least until ψ holds;
Temporal	ϕ R ("release") ψ	ϕ has to hold until and including ψ holds; If ψ never holds, ϕ has to hold forever;
Temporal	ϕ W ("weak-until") ψ	ϕ has to hold until ψ holds; If ψ never holds, ϕ has to hold forever;
Logical	$\phi \wedge \psi$	ϕ and ψ have to hold together;
Logical	$\neg\phi$	if ϕ does not hold, $\neg\phi$ holds;
Logical	$\phi \rightarrow \psi$	if ϕ holds, it implies that ψ will also hold;
Logical	$\phi \leftrightarrow \psi$	if ϕ holds, it implies that ψ will also hold, and vice versa;
Logical	$true$	the formula that always holds;
Logical	$false$	the formula that never holds;

Table 2.1 Temporal and logical operators in LTL

Semantics

To evaluate whether a LTL formula is satisfied or not, a structure is needed, often called an interpretation. The LTL formula alone is not evaluable regarding it's satisfiability. We use Kripke structures as interpretation. The LTL formula is satisfied if a specific sequence of APs hold (evaluate to true). This sequence is a ω-word on a trace of the Kripke structure. An ω-word is an infinite sequence of symbols, in our case an infinite sequence of APs. If the formula evaluates to true, we say: "the ω-word on the interpretation satisfies the formula" [33]; The ω-word is indexed, meaning that the position i is related to an AP at the $\omega[i]$-word. This is how time can be visualized in a Kripke structure, also the past [35].

The advantage of LTL is its temporal expressiveness. It is possible to describe certain criteria for a system that have to hold, which are also called properties. Properties can be classified. Subsequently, the two most important property classes of LTL are explained [34] [36].

Safety: "Something bad will not happen." An example for such a property looks like this: $\square \neg$(reactor_temp > 5000)

Liveness: "Something good will happen infinitely often." An example for such a property looks like this: $\square$(start $\rightarrow \lozenge$ end)

2.7.3 Computation Tree Logic

Computation Tree Logic (CTL) is a close relative of the LTL, where both are a subset of CTL*. CTL is a branching-time logic, it investigates the system model (Kripke structure) as a computational tree with the initial state as the root node and all possible transitions of that state and of the subsequent states as branches. This visualizes all potential paths in a system. CTL offers the possibility to define properties that have to hold on specific paths of the structure. To enable that, quantifiers have to be attached to the properties. These quantifiers are A and E. The A ("All") quantifier defines that the attached property has to hold on every path of the computational tree from the current state. The E ("Exists") quantifier determines that there is at least one path in the computational tree from the current state on which the attached property has to hold [36].

The A quantifier can be seen as integrated by default in the LTL, meaning that all properties defined in LTL have the A quantifier attached to it. The E quantifier is not expressible in LTL. It can be seen as a weakening of the

property, so that not in every traversal of the Kripke structure it has to hold, but in at least one [36]. This is important for reactive systems when a model checking approach is used to verify the entire system. Figure 2.8 shows a simple example for a reactive system. The system receives in $S_{GetInput}$ some input data of a user. Depending on the evaluation of the input data, it transits to $S_{GoodInput}$ or to $S_{BadInput}$ and terminates in $S_{Finished}$. If we want to verify that at least in one case the transition to $S_{BadInput}$ is taken, we can not express this with LTL, because the property has to hold always on all paths. The correct notation in CTL for this case would be: $E(S_{GetInput} \bigcirc S_{BadInput})$

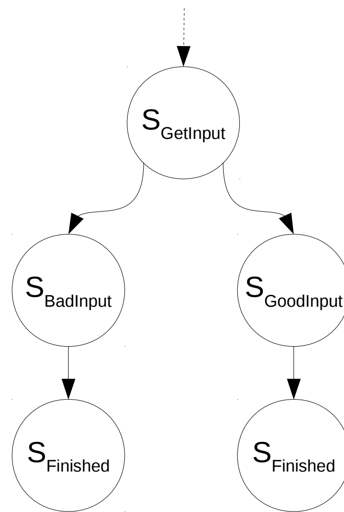

Fig. 2.8 Kripke model to explain expressiveness of CTL compared to LTL.

2.8 Related Work

Discussions with colleagues at the DLR regarding *Model-Based Development (MBD)* revealed that it is in use in the industry, but primarily in the mission phases 0, A, B and C. There, MBD methods are used to define mission goals and requirements for an entire mission, like done with the Virtual Satellite project [37]. Or the definition of non-behavioural software structures, conducted in the ATON project [38]. Additional investigations on MBT for OBSW of satellites showed that some specific components get verified using a MBT approach (like the AOCS). But the approaches are tailored specifically for the component and do not match the requirements for our test-framework. Therefore, an investigation of related industries and industry generic methods in the field of MBT is done.

Nan Li and Jeff Offutt tackled the problem of the transformation of abstract tests into executable concrete tests. They identified four major issues in the field of MBT: "(1) building the test model, (2) using test criteria and algorithms to generate abstract tests from models, (3) transforming abstract tests to concrete tests and (4) generating test oracles in concrete tests" [39]. To overcome these problems, they invented STALE. STALE is a framework with an equally named language. The STALE language creates mappings between the SUT and the abstract tests. The SUT is expected to be written in an OO language. To achieve the mapping, the mapping object has a unique name, the type of an element of the SUT and the name of the element. In case of a mapping to a Class type, the name of the object has to be given too. In the body of the mapping object, the test designer is able to set the referenced OO object in an initial state, before the constraint (test case) that has to hold is written. These test cases are translated in executable code by the framework. An evaluation of the efficiency of the framework revealed that 29,6% less time is needed to generate the executable tests, compared to the manual method [39].

In the paper *Integrated model-based approach and test framework for embedded systems* the authors present a test-framework to test *Resource-Constrained Real-Time Embedded Systems (RC-RTES)*. The user can define a non-behavioural model of the SUT in the UML language, which is transformed in an executable application for a RC-RTES via a code generator. The UML model is linked to a test model, which is part of the test framework. The test model is called *proxy test model* and can be used to define test cases with input and expected output for specific components of the UML model. The test input data is transformed into executable stimuli and transmitted via a *Target Debugger* to the location of the implemented application on the RC-RTES. In return, the result of the execution is transmitted back to the *Target Debugger* and resolved by the test-framework to compare it to the expected output. The verdict of that comparison can then be investigated in another UML diagram [40].

Shaoying Liu's paper describes the challenges of *Automatic Specification-Based Testing (ASBT)* [41]. *Specification-Based Testing (SBT)* is the creation of test cases with the specification as the sole input. The structure or behaviour of the SUT remains unused. This method of test case creation is often also referred as "black-box testing" or "model-based testing". ASBT is basically the same, but every task in the entire testing process ought to be automatized.

The **specification language** to describe the specifications can be either formal or informal. Informal languages are hard to use for ASBT due to their ambiguity and the lack of a well-defined structure. Formal languages are more

suitable for ASBT purposes, as they can be created with a strongly structured domain emphasis. That facilitates the extraction of information, which is necessary for the latter steps. Additionally, interface gaps have to be considered between the specification language and the test execution platform as well as the SUT [41].

The **method for generating adequate test sets** is difficult to implement. Coverage criteria or termination criteria can be used to determine the test case sets. But the obtaining of the required information that terminates the test case generation process is difficult. Model checking approaches gained attention over the last years [41].

The **translation between the abstract data and the concrete data** is challenging. Mostly, the abstract test cases of the specification can not be directly used in a test execution platform. They need to be translated into concrete test cases. The same counts for the expected abstract and concrete test output (oracles) [41].

Important to bear in mind when **analysing test results to determine bugs** is, that the oracles need to be at least defined for any output. To determine the correctness of an output, the output must be traceable, otherwise there will be always a bug detection. Only defined oracles are outputs with "don’t cares" [41].

Debugging is the localisation of a bug in the software. The challenge is to trace a bug back from the SUT to the specification. When testing, the SUT is traversed in paths. In case of a bug it is needed to determine the exact state in which the bug occurred and not only the initial and the terminal state of the traversed path [41].

Integration testing is a huge process in overall testing. In ASBT a trade-off between the completeness of the integration test and the provided information in the specification has to be accepted. This is due to the fact that not every test case produces a traceable output. One solution is to add extra pre- and post-conditions in the specification for the execution of a test case, but this increases the workload on the specification writer. Also the incompleteness of testing has to be more or less accepted in ASBT, as the tests always traverse paths from a specific initial state to a terminal state. Testing on all initial states is almost impossible for huge SUTs [41].

Chapter 3

Project's Ethic Responsibility

The chapter introduces to the engineering ethics with the famous O-ring failure, which led to the Space Shuttle Challenger disaster. Following, the Code of Ethics for Engineers with its Fundamental Canons is elaborated and subsequently applied to the test-framework and its development.

3.1 Space Shuttle Challenger Disaster

There are several historical examples in which wilful ignorance of ethical standards lead to a catastrophic outcome, sometimes even lethal for humans. One of the most famous ones is the malfunction of an O-ring sealing, which lead to the Space Shuttle Challenger disaster in 1986.

The catastrophe occurred due to a fissure of at least one of the O-ring seals of the solid rocket boosters attached to the space shuttle. These sealings were obstructed between the four segments of the booster. The low temperatures of the night before the launch decreased the flexibility of the sealing material, leading to a brittle failure, which couldn't stand the sudden calefaction while the launch. The hot fuel-gas leaked out of the booster and eventually also got inflamed by the nozzle, although this could not be clarified entirely. The fuel-gas blow-out had contact to the adjacent external tank of the space shuttle, leading to a structural failure. After 73 seconds, the entire spacecraft broke apart due to aerodynamic forces, killing all seven crew members [42, p: 339-ff].

The subsequent investigation of the accident, regarding the cause for the disaster, was conducted by the Rogers Commission [43]. The result of the investigation of the commission was, that the design of the O-ring was insufficient, considering NASA's requirements for low temperature environments. Roger Boisjoly and other engineers of Morton Thiokol (manufacturer of the boosters) were aware of this insufficiency, and informed their management in the night

before the launch. The management of Morton Thiokol briefed the NASA and quickly established the Seal Task Team to investigate the risk. Though, the managers of Morton Thiokol under managed the Seal Task Team and overruled the recommendation of Roger Boisjoly not to launch the space shuttle. Against all warnings, the NASA accepted the recommendation of Morton Thiokol to launch and initiated the catastrophe [42, p: 339-ff].

Roger Boisjoly tried to follow the first Fundamental Canon of the Code of Ethics for Engineers, but got overruled by his management. The managers did not want to delay the start, due to financial and reputation concerns. The NASA was aware of a potential risk and agreed on it. In the end, the Rogers Commission blamed the NASA as the main culprit for the disaster [43]. If the management of the NASA or Morton Thiokol would have obeyed the Code of Ethics for Engineers, seven lives and billions of US Dollars might have been saved.

3.2 Code of Ethics for Engineers

The Code of Ethics for Engineers is one of many existing ethical guidelines (compare: IEEE Code of Ethics [45]), established by the National Society of Professional Engineers [44]. The code consists of three basic sections: the Fundamental Canons, the Rules of Practice and the Professional Obligations. The goal of the codex is to provide basic rules with explanatory use-cases, to help engineers taking ethically as well as morally correct decisions in average and exceptional situations.

The Fundamental Canons

Obeying the Fundamental Canons should help to remain in a state of ethical correctness. They serve as the basis for the entire codex and have to be taken as is. The Fundamental Canons are [44]:

1. "*Hold paramount the safety, health, and welfare of the public.*"
2. "*Perform services only in areas of their competence.*"
3. "*Issue public statements only in an objective and truthful manner.*"
4. "*Act for each employer or client as faithful agents of trustees.*"
5. "*Avoid deceptive acts.*"

6. "*Conduct themselves honourably. responsibly, ethically, and lawfully so as to enhance the honour, reputation, and usefulness of the profession.*"

The Rules of Practice

This section of the Code of Ethics for Engineers provides the explanation of the Fundamental Canons. The single cannons are enriched by potential scenarios which could occur in an engineers life. Following, an example scenario of the Rules of Practice is cited, which targets the Fundamental Canon number one:

"*If engineers' judgement is overruled under circumstances that endanger life or property, they shall notify their employer or client and such other authority as may be appropriate.*" [44]

The Professional Obligations

Compared to the Fundamental Canons, which can be seen as strict rules, the obligations cover the moral aspect of the codex, with the focus on social interactions. That does not mean the Professional Obligations are less important, but breaking those obligations is in general not as harmful as breaking one of the canons. An example for an obligation is:

"*Engineers shall at all times strive to serve the public interest.*" [44]

3.3 Ethical Aspects of the Project

Under consideration of the Fundamental Canons, the developer of the system has to bear the following ethical rules particularly in mind:

- **Hold paramount the safety:** The test-framework evaluates the correct functionality of parts of the OBSW, which later will run on the specific hardware. In case of a flawed implementation of the test-framework, test results might not forecast the actual OBSW behaviour. If this is the case, hardware in the loop tests could lead to a severe damage of the hardware or even worse of the operator. Every user has to be aware that there might be an error in the test-framework, but the developer has the duty to reduce this risk as much as possible. Therefore, the conduction of meticulous and precise actions during the development of the design

and the implementation are of importance. Furthermore, documentations and operational manuals have to be provided to allow other engineers to detect potential risks and to inform the test operator about the capabilities and the handling of the test-framework.

- **Perform services only in the area of your competences:** Here, the admission of personal knowledge limitations are in the focus. The testing of software grew strongly over the last years. New technologies like symbolic execution or model checking became usable within this field. Additionally, the creation of an automatic testing platform on the basis of DSLs is not trivial. The degree project takes place in the aerospace environment, which has its own rules and systems. For field-external developers, they are hard to catch on, especially in the short amount of time provided. The developer has to overcome theses barriers. The solution is the discussion with experts in the various fields, to profit of their knowledge. Being aware that one's personal knowledge is limited, seeking for help is the way to hold that Fundamental Canon.

- **Issue public statements only in an objective and truthful manner:** This canon is important for all presentations as well as documents, like this report, developed within the project. This counts for internal as well as external documents. Telling the untruth creates risks for the colleges, for the employing company and for oneself. This in combination with objectiveness, is one of the bases for a successful project, which can be used to develop further.

- **Conduct themselves honourable and ethically to enhance the reputation of the profession:** The test-framework is planned to be used. Therefore, it's quality has to be sufficient and it has to be created under the consideration of ethical aspects. This is important, because the DLR as well as the colleges are inferred by the result of the project. In case the ethic code is not obeyed and externals are aware of that, the reputation of the colleges and the DLR might suffer. Hence, the goal is to increase their reputation by the delivery of a honourable result.

Ethical codex conformity ensures a long-term success everyone can benefit from. Sometimes these ethical rules seem to slow you down in your personal development, but this impression is wrong. They actually enforce a sustainable and continuous development, which in the long run helps you to reach your desired goals faster. Ethic codes are not an invention of engineers, they

evolved in the field of social science and simply got adapted to the engineering profession. There is an option to replace all the Fundamental Canons by one and still fulfil the Code of Ethics for Engineers or any other ethical codex: Live the Categorical Imperative.

Chapter 4

Analysis and Requirements

This chapter displays three types of requirements which determine the final design of the test-framework. The general requirements focus on the handling of the test-framework by the domain expert and the programmer. The protocol requirements are gained via an analysis of the Packet Utilization Standard. They consist of the packet structure requirements and the behavioural expressiveness requirements. The third are additional temporal requirements which target specific temporal behaviour.

4.1 General Requirements

The general requirements are based on the experience of the employees at the Institute for Software and Simulation of the DLR, regarding the collaboration with domain experts of various scientific fields. The general requirements are denoted in the subsequent Table 4.1.

Enum	Requirement	Motivation
1.	Domain expert as test designer	The domain expert is the test designer, who is using his or her experience and domain knowledge to define the tests.

2.	Simple to use tool	Parts of the framework must be operated by non-programmers, which are not used to IDEs like Eclipse. Keeping the tool facile to utilise reduces the threat of rejection.
3.	Steep learning curve via repetition	Repetitive patterns in the design of the tool help to accelerate the learning how to operate the same.
4.	Date-centric design	The DDL describes the structure and the interfaces with their parameters. This must be pursued by the test-framework.
5.	As automatic as possible	The more the framework is automated, the less work will remain for the programmers and the domain experts. Also the threat of oversight is reduced.
6.	Enable test-driven development	Tests have to be written before the OBSW source code.
7.	All outputs in XML	The outputs have to be in XML format to be able to automatically generate a test documentation in a standardized way, using Maven or other tools.
8.	Logs and reports	Logs display all TC and TM transmissions from and to the OBSW. Reports denote the result of a test case.

9.	Exploit re-usability	Model components should be re-usable in the model as well as in other mission models to reduce redundancy.
10.	Functional testing	The test characteristic of the test-framework is to test the functionality of the OBSW.
11.	Manual test creation	The domain expert has to be able to define test cases manually, based on his or her domain specific experience.

Table 4.1 General requirements of the test-framework

4.2 Protocol Requirements

The protocol requirements are elaborated by an investigation of the PUS. In focus is the TC/TM packet structure and the expected specific behaviour determined by the standard. It is not requested to fully implement the standard, but core parts of it. This is due to the fact that the standard always gets tailored according to the mission requirements and therefore varies [17].

4.2.1 Package Structure

Figures 4.1 and 4.2 visualize the structure of the packets defined in PUS.

In the test-framework, TC and TM packages can have the same layout, as most of the packet type specific flags refer to a specific behaviour which will not be implemented. The resulting structure of the packets is visible in Figure 4.3. The structure of the packets has been simplified. Beginning with the *Packet Header*, the *Type* denotes whether it is a TC or a TM packet (TC = 1). The *APID* remains as in the PUS packet structures, but the *SERID* and the *SUBSERID*, which are normally located in the *Packet Data Field*, are extracted and added to the *Packet Header*. The *Packet Sequence Count* remains, and helps to identify the

Packet Header (48 Bits)							Packet Data Field (variable)			
Packet ID				Packet Sequence Control		Packet Length	Data Field Header (Optional)	Application Data	Spare	Packet Error Control
Version Number (=0)	Type (=1)	Data Field Header Flag	Application Process ID	Sequence Flags	Sequence Count					
3	1	1	11	2	14					
16				16		16	Variable	Variable	Variable	16

Fig. 4.1 Telecommand packet structure

Packet Header (48 Bits)							Packet Data Field (variable)			
Packet ID				Packet Sequence Control		Packet Length	Data Field Header (Optional)	Source Data	Spare (Optional)	Packet Error Control
Version Number (=0)	Type (=0)	Data Field Header Flag	Application Process ID	Grouping Flags	Source Sequence Count					
3	1	1	11	2	14					
16				16		16	Variable	Variable	Variable	16

Fig. 4.2 Telemetry packet structure

packet, to enable the comparison with the Oracle packets. The *Packet Length* determines the total length of the *Application/Source Data* of the *Packet Data Field*.

Packet Header (57 Bits)						Packet Data Field (variable)
Packet ID				Sequence Count	Packet Length	Application / Source Data
Type	APID	SERID	SUB-SERID			
8	8	8	8	16	16	variable

Fig. 4.3 Packet structure of the TC/TM packet of the test-framework

4.2.2 Behavioural Expressiveness Requirements

TM verification

The expected basic scenario is that the test-framework sends a TC packet to the OBSW and expects a TM packet in return. The returning TM packets is

compared to an expected output, the Oracle packet.

TC/TM sequences
TM verifications for each TC are not necessary. In PUS there is also the possibility to send a sequence of TC packets to the satellite, expecting a TM in return after the execution of the entire sequence. This is very important in case of an error event, when ground tries to find a workaround sending a sequence of low-level commands, like: "switch relay" or "load register". In addition, a sequence of TM packets, created by the OBSW in return, has to be possible too.

Procedures
Testing is incomplete verification and can be seen as traversing paths through the OBSW/SUT. The test designer has to be able to concatenate multiple states to build a chain of states and therefore a trace in the OBSW, represented as a Kripke structure. Procedures are concatenated TM verifications or/and TC/TM sequences.

Periodic TM packets
The PUS defines two types of TM packet transmissions, housekeeping and event-based. Event-based are aperiodic transmission of TM packets, triggered by events in the system (for example by an arriving TC packet). Housekeeping packets are periodic TM packets, that are continuously transmitted. The event-based transmissions are covered by the prior mentioned TM verification. The domain expert has to be able to define the housekeeping packets and their expected interval arrival time.

Missing TM packets
Due to latency reasons of the link between the base station and the orbiter, the tracing of missing packets is shifted from the transport layer to the application layer (there is no Transmission Control Protocol used for the communication). The in PUS defined Sequence Count is used to identify missing packets. The test-framework has to wait for a configurable amount of time for all TM packets to arrive and denote missing packets in a report.

Table 4.2 summarizes the gained requirements of the behavioural aspect of the PUS for the test-framework.

Enum	Expected behaviour	Explanation
1.	TM verification	There must be an option to define expected TM packets in response to certain TC packets.
2.	TC/TM sequences	Concatenations of TC/TM packets must be possible.
3.	Procedures	Concatenations of TM verifications and/or TC/TM sequences must be possible.
4.	Periodic TM packets	Periodic housekeeping TM packets have to be definable.
5.	Missing TM packets	Missing TM packets have to be reported.

Table 4.2 Expected behavioural expressiveness of the test-framework

4.3 Additional Temporal Requirements

The additional temporal requirements target basic testing scenarios which are not covered by the PUS.

Enum	Temporal behaviour	Explanation
1.	Always	The tester must be allowed to define a TM packet that always arrives as defined.
2.	Never	The test designer must be able to define TM packets which should never arrive.
3.	Until	The test designer must be allowed to define a property that has to hold until a certain TM packet arrives (state is reached).

Table 4.3 Additional temporal requirements of the test-framework

Chapter 5

Design

In this chapter the design of the test-framework, based on the defined requirements of the Analysis and Requirements Chapter is explained. The design elaboration follows a top-down approach. In the first section, generic top-level design considerations are elaborated. Based on that, the model-based taxonomy is applied, to determine the technical capabilities of the components of the test-framework. The subsequent section uses the output of the prior ones to explain the actual top-level design with its three main parts. Finally, each part is detailed in a distinct section.

5.1 Generic Top-Level Design Considerations

The generic top-level design considerations section targets universal design questions which are of importance for any test-framework. The test dimensions are defined and the used approach for the test-framework elaborated. Both considerations are based on the prior in the Analysis and Requirements Chapter (4) defined requirements.

5.1.1 Test Dimensions

The test dimensions cluster decisions on test goals, test scope and test abstraction. They are visualized in a coordinate system in Figure 5.1. The outcome of the application of the requirements on this coordinate system serves as a first basis for the test-framework.

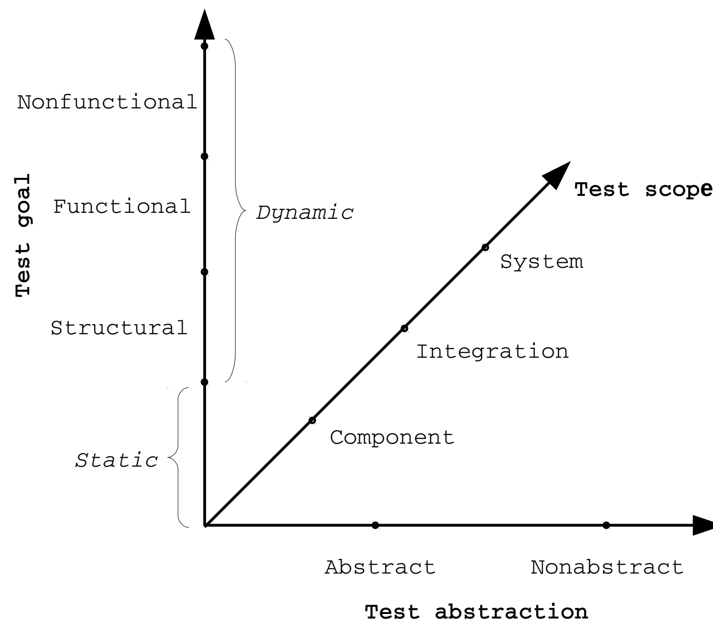

Fig. 5.1 Test dimensions. Source: [46, p:5]

Test Goal

The test goals depend on how the tests are conducted. In the Background Chapter, testing is defined as a dynamic process, excluding static means like code reviews (2.2). Therefore, our test goals are the ones achievable by dynamic verification.

Structural tests cover the structure of the SUT. They are also called white box test or glass box tests. To derive test cases for such a test, insights on the structure of the SUT must be given [46, p:6]. As the test-framework has to enable test-driven development, the SUT is coded after the tests have been written. Therefore, a direct investigation of the structure of the SUT is not possible. But the model defined with the DDL describes the structure of the final SUT with sufficient detail to investigate the data-flow of the packets in the structure. Hence, **structural tests** are implemented and defined as the first test goal.

The second test goal is based on the requirements of the test-framework, which demand **functional tests**. This is in our case the correct reaction of the OBSW in the form of expected TM packets on stimuli in the form of specific TC packets. Non-functional goals like reliability, scalability or maintainability are not covered by the requirements and thus not a test goal of the test-framework.

Test Scope

The test scope defines the granularity of the tests, reflected in the granularity of the SUT [46, p:6]. The test scope is also called the test level. In the coordinate system the granularity decreases when veering away from the center. To define the test scope, the available granularity has to be considered.

The component scope is the granularity which is necessary for unit tests [46, p:6]. The model of the system, described with the DDL, does not have that granularity. And, as already mentioned, the SUT is at this stage not yet available. Therefore, components are out of the test scope.

The **integration scope** considers subsystems and the **system scope** the entire system [46, p:6]. For these test scopes, the granularity of the model is sufficient. They are the target scopes of the test-framework.

There are additional higher scopes like the regression or the acceptance scope. In the space industry, these test scopes are covered by the Acceptance Review and the Operational Readiness Review milestones in the mission phase D, as explained in the Introduction Chapter (1.2).

Test Abstraction

The test abstraction defines the gap between the final executable test and the test specification. The higher the abstraction, the higher the readability and the re-usability of a test specification. But the test specification must provide enough data to be able to derive the executable test cases of it [46, p:7].

In the test abstraction dimension, the requirements regarding the steep learning curve, the simplicity of the usage and the domain expert as the test designer are in focus. The abstraction must be high enough to reach such requirements and low enough to derive the executable test cases of the test specifications. The exact position on the coordinate system is blurry, but the goal is to push it as far as possible towards the center, to keep it **as abstract as possible**.

5.1.2 Test-framework Approach

The test-framework has to dynamically verify the behaviour of the SUT regarding it's expected behaviour by selecting test cases of the infinite test case domain. Test-driven development has to be enabled, by a specification-based approach of the test case generation, conducted by the domain expert. Additionally, the framework ought to be as automatic as possible and re-usability is in the focus. The produced output has to be in XML format, which has to hold

for the logs as well as for the reports. Based on these requirements, the approach for the test-framework is selected, which are detailed in a distinct section in the Background Chapter (2.3). Pure manual testing is not investigated, as high automatism is requested.

For the rest of the document SUT and OBSW can be seen as equivalent expressions for the same thing. They are interchangeable, whereas the expression OBSW ought to direct the attention of the reader in the direction: software construct for satellites. And the expression SUT should help to identify it as an appended part of the test-framework.

Capture and Replay

The capture and reply technology allows a good comparison between different versions of the SUT, backtracking failures to a specific change in the SUT. But it is also very sensitive on API changes. This can lead to a huge amount of incorrect failure detections between different OBSW versions. The encapsulation of the SUT requires OBSW while writing the tests, which is by the requirements (enable test-driven development) not given. These counterarguments prohibit an implementation following capture and replay.

Scripts

Scripts are simple to implement and help to automatize the testing process. But the test designer needs a deep understanding of the OBSW, which is not given, as the test designer is the domain expert. Problematic is also the amount of test scripts which would be necessary to test such a huge construct like an OBSW for satellites. Additionally, the SUT must be given when writing the test to incorporate parts of it in the scripts, which is again not the case. These reasons forbid a concept based on scripts for the test-framework.

Data-driven

The data-driven approach with scripts enables better re-usability due to the data separation, compared to basic scripts. Whereas the implementation is as simple as in basic scripts. But the redundancy is still relatively high, because the test scripts also contain the functional implementation of the SUT, which is not available. The result of this investigation is that the data-driven approach with scripts can not be exploited for the test-framework.

Keyword-driven

The keyword-driven approach is based on functional decomposition by the insertion of an abstraction layer. The used action words result in an increased readability and understandability, which allows also non-programmers to design test cases. The action words reduce the redundancy compared to a data-driven approach with scripts. Contradictory is the expected technical expertise of the test case designer, which is in general a negative aspect. But in our case the domain expert has this knowledge, which transforms this disadvantage to an advantage. A real disadvantage is the increased complexity of such a framework and the inherent increased required management support. Also the coverage criteria has to be developed manually, which is costly.

Model-based

The model-based approach belongs to the keyword-driven approach family. It owns the same advantages and disadvantages, with one major difference. The coverage requirements are easier to implement and allow an automatic evaluation. This increases the overall coverage over time of the tests. A coverage criteria determines the point in time for the test case generation termination. This is needed when automatic test case generation is requested. Therefore, the model-based approach is also referred as an "intelligent framework". A disadvantage of the model-based approach which often leads to a rejection, is the necessity of a model of the SUT. But in our case, the model is already provided by an instantiation of the DDL. When balancing the advantages and the disadvantages of all approaches in context to the requirements, a model-based approach is the most promising solution.

5.1.3 Different Model-based Approaches

In the Background Chapter, the four major MBT approaches have been elaborated (2.4.1). Subsequently, these insights are applied to the requirements.

The **generation of test data from a domain model** approach is only walking half the way, as the correctness of the tests cannot be evaluated automatically due to the lack of oracles. Additionally, it is for our case not possible to test all possible input data, as this would create too many test cases. And the focus is clearly on specific behaviour of the OBSW, related to specific input.

The **generation of test cases from an environment model** approach would solve the problem of too many implausible test input data and therefore test cases, as the data is obtained by an environment model. But currently,

there is no environment model available and the problem of the lacking oracles also remains here.

The **generation of test cases with oracles from a behaviour model** approach allows the creation of specific input data and correlating oracles of a behavioural model. The provided system model by the DDL is a non-behavioural model. To be able to use this approach an additional behaviour model or a modification of the system model is requested. A modification of the system model is not possible, as the DDL has to remain fully functional, also when standing alone. Simply copying the DDL and building up on it to create an implementation which allows the definition of behaviour is also not recommended, as two modelling softwares of one system would have to be maintained in parallel. An additional problem is that modelling behaviour is difficult for a domain expert with insufficient programming skills. Basic constructs in programs have to be known. Learning them just to be able to operate the test-framework is against the requirements of the steep learning curve and the "simple to use" paradigm.

The **generation of test scripts from abstract tests** approach targets the creation of executable test cases from an abstract test case definition by the usage of the system model as interface and structure information source. This is matching our requirements. The DDL describes the routing behaviour and the data structures of the SUT. Using only this as input to generate test cases would offer a limited test coverage, capping only the structural test goal. But the entire functionality of the OBSW ought to be tested too. Therefore an additional test model has to be created which allows a precise definition of input and expected output data (oracles) in the form of abstract test cases.

5.2 Application of the MBT Taxonomy

To derive the functionality of the components of the test-framework, the MBT taxonomy is used. The taxonomy with various options is elaborated in the Background Chapter (2.4.2).

5.2.1 Model Class

MBT Basis

The "generation of test scripts from abstract tests" approach is selected for the test-framework. The *system model*, instantiated with the DDL, provides enough information for structural tests, but not enough for functional tests. Hence, an additional test model is required for the description of the expected

behaviour of the OBSW. To exploit the re-usability, which is a defined requirement, the two models are linked. This allows the test model to reuse data structures and interfaces of the system model. Therefore, a **coupled system and test model** is selected as testing basis.

Properties

The properties are based on the same requirements as the prior explained test abstraction. The system model and the test model have to follow an "simple to use" paradigm with a steep learning curve with the domain expert as the main user. Therefore, the layout of the modelling language to describe the tests has to be traceable and somehow related to human language. The gap between the existing DDL layout and the test modelling language should be kept as small as possible to steepen the learning curve. The handling of the framework by the domain expert has to be easy and self explaining.

5.2.2 Test Generation Class

Test Selection Criteria

Remember that when the test cases are generated only the coupled system and test model is available, not the SUT. A mutation of the system model is not recommended because of the lack of sufficient behavioural information for the functionality testing. Mutating the test model to determine input equivalence classes would be feasible, but the input data is infinite and a sufficient test coverage criteria hard to reach. Additionally, the later evaluation on the correctness of the test case's output would be an enormous manual task for the domain expert, due to the shire amount of test cases. The usage of data coverage to generate tests creates too many test cases, also if the input data is limited. And the later evaluation, as prior described in test model mutation, is equivalently arduous. The requirements coverage option is too informal and is the current method to create integration and system tests for OBSW. The task of the test-framework is to overcome this knowledge exchange process. Also random and stochastic test case generation is not applicable due to the infinite potential input data and the difficulties to determine a sufficient coverage.

Therefore, the test-framework follows a mixture of two options for the test selection criteria: the **test case specification** and the **fault-based**. The domain expert is the designer of abstract test cases. His or her knowledge on the expected behaviour of the system can be exploited by the **test case**

specification option. Additionally, his or her insights on potential faults can be used for the **fault-based** option.

Problematic is the coverage criteria, as **test case specification** and **fault-based** do not have a measurable coverage criteria. The test coverage is reached when the domain expert states the amount of tests as sufficient. To help him or her not to oversee test cases, an additional **structural model coverage** is used. This is done by the evaluation on the data-flow in the system model. The domain expert receives a warning in case not all possible data-flow transitions of the system model are covered by at least one test case.

Test Generation Technology

A manual generation of the test cases is not conformable with the required automatism. Additionally, this is not closing the gap between the programmer and the domain expert, as the programmer is the one who would have to write the executable test cases, based on the specification of the domain expert. This is again the current knowledge exchange process. Additionally, a random generation and a graph search algorithm is not combinable with the defined test selection criteria. Model checkers and theorem provers are good means for the test case generation in case there is a huge amount of potential test input data. In the case of this framework, the input is already determined precisely by the domain expert. This makes the usage of such technologies senseless.

The for this project selected test generation technology is an **automatic offline** approach. All abstract test cases in the test model are translated into executable test cases. These test cases cover the functional tests. Invisible for the domain expert, structural test cases are attached to every of him or her defined abstract test case. The generation of the test cases is conducted before the test run begins and is therefore **offline**.

Result of the Generation

The output of the test generator of this project is **executable code**, as the goal is to transfer the knowledge over the gap between the domain expert to the programmer with the help of the models.

5.2.3 Test Execution Class

Execution Options

The primary execution option is the **SiL**, because this is the stage in which the functionality of the OBSW can be tested for the first time in the development process of the satellite. But also **PiL** and the **HiL** execution options are possible. To enable that, an interface between the computer on which the test-framework is executed and the on-board computer of the satellite has to be created. This is clearly in the focus of the employees at the DLR for the test-framework, but exceeds the extend of this thesis. Vehicle tests are generally not conducted in satellite development, especially not in scientific satellite development. Therefore, this stage is left out.

The test-framework is basically **reactive**, but with certain limitations. Some "waiting" test cases get executed if structural tests of prior executed test cases are successful. But the evaluation of the functional tests has no impact on the "waiting" test case. That means if the functional tests of a test case fail, the structural tests of a test case pass and another test case is waiting on the test case, the waiting test case will be executed. This is important to reduce the pressure on the test-framework by forcing the programmer to check every functional test case evaluation manually. It is necessary because the system or the test model of the domain expert might have errors, which could remain undetected when the test-framework automatically states test cases as passed.

Also the **generation of logs** is implemented. There are two types of logs. A transmission log, in which all transmitted TC and TM packets are chronological listed. And an unresolved packet transmission log, which protocols arriving TM packets form the OBSW at the test-framework which are not expected or unknown.

5.2.4 Test Evaluation Class

Specification

The test-framework uses two options of this category. The **reference signal-based specification** is used to directly compare the parameters of the packets with the specified oracles. This is used because the derivable features of the TM packets of the SUT are not capable to evaluate the correctness of the behaviour of the OBSW. And the **test evaluation definition** option is applied to determine the specific oracle and the time when to evaluate it against the SUT's output.

Technology

The evaluation of the test cases is conducted in fully **automatic** manner, **creating reports** in XML. This is possible because the used test generation technology is automatic, which also covers the generation of the test oracles. These test oracles are needed for the evaluation at this stage.

Additionally, the evaluation process is conducted in an **online** fashion. This is important as the tests might run for hours and the programmer needs an immediate feedback on the correctness of the OBSW. The worst case scenario would be an offline evaluation of a long time test with the result that in the first seconds a simple OBSW failure destroyed the entire test run's expressiveness. In case of an offline approach, this would be only visible in the final report.

5.3 Top-level Design

In this section the top-level design of the test-framework is elaborated and the workflow with the test-framework explained. Figure 5.2 displays the design overview of the test-framework. It consists of three main components: the *Data-Description Language (DDL)*, the *Data-Testing Language (DTL)* and the *Test Execution & Evaluation Platform (TEEP)*.

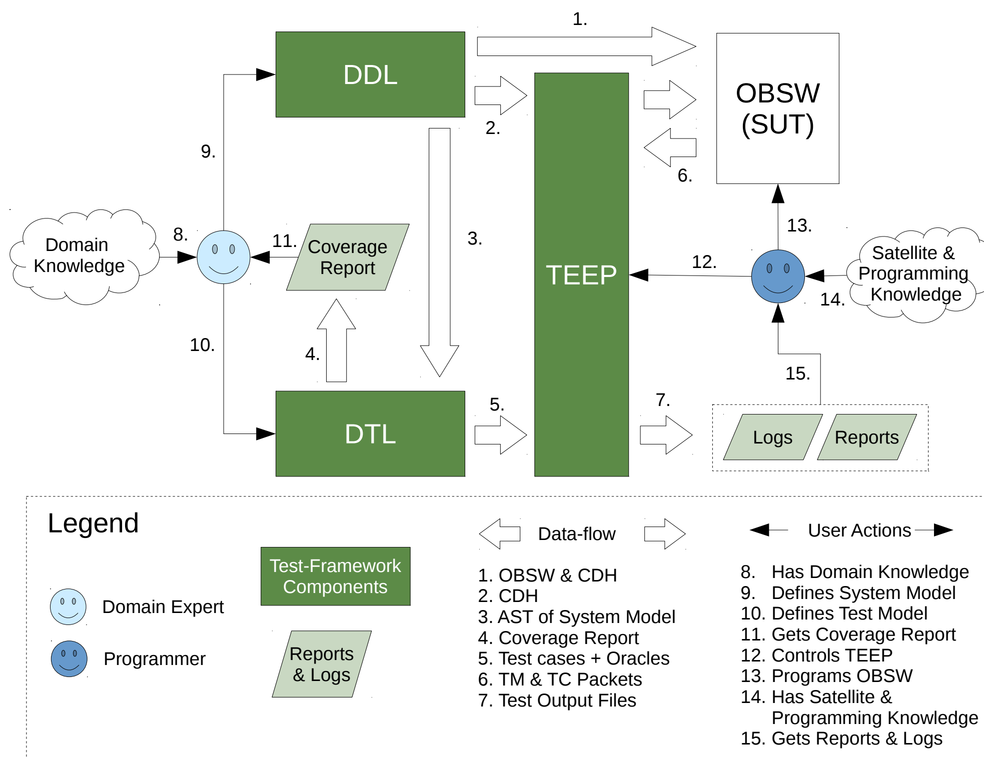

Fig. 5.2 Overview of the test-framework design and workflow.

5.3.1 Data Description Language

The DDL is a modelling language to define and instantiate the system model. The domain expert, with his or her specific domain knowledge (8.), uses the DDL in a specific editor to define the CDH subsystem structure and the TC & TM packets (9.) of the OBSW. One output of the DDL are code stubs for the OBSW, representing the CDH subsystem and the structure of the OBSW (1.). Additionally, the CDH is provided for the TEEP (2.). The third output is the AST of the instantiated model (3.). The layout of the DDL is following the defined properties of the MBT taxonomy's Model Class. The same counts for the subsequent DTL.

5.3.2 Data Testing Language

The DTL is also a modelling language, used to define the test model. The AST of the system model is accessible within the editor of the DTL. This is the linking between the two models and therefore the coupled system and test model. The domain expert initializes the in the DDL defined TC & TM packet's parameters, to create the *Test TeleCommand (TTC)* and the *Oracle* packets (10.). Subsequently the test cases get defined, which are potential scenarios for the domain expert's component (10.). The test case definition, as well as the TTC and Oracle packet initialization are based on the specific insight of the domain expert and follow the test case specification and fault-based criteria of the MBT taxonomy's Test Generation Class. Feedback on the completeness of the test suite is provided by the coverage report, which determines the structural coverage of the system model (4. & 11.). When the coverage of the test suite is sufficient, depending on the coverage report and the opinion of the domain expert, his or her work is done. A code generator automatically translates the test cases with the TTC and the Oracle packets into executable C++ classes and integrates them in the Test Execution and Evaluation platform (5.). Additional structural tests are integrated, hidden from the domain experts eye. This covers the rest of the Test Generation Class categories.

5.3.3 Test Execution & Evaluation Platform

The TEEP is used to execute and evaluate the test cases regarding the expected behaviour and the actual behaviour of the attached OBSW (SiL). It covers the Test Execution and Evaluation Classes of the MBT taxonomy. The programmer is in control of the TEEP (12.) and programs the OBSW (13.), based on his or her satellite OBSW and programming knowledge (14.). The

OBSW receives stimuli in the form of TTC packets and returns TM packets to the TEEP (6.). The order of the stimuli and the Oracle packets, which are compared to the arriving TM packets, follow the generated test case scenario of the domain expert. The TEEP creates log entries of the transmission of the packets and report entries with the test evaluation results (7.). The reports and logs are primarily provided for the programmer (15.). In case the programmer discovers a miss-design by the domain expert, he or she is able to discuss this with him or her on the basis of the system and test model, the reports and the logs.

5.4 Data Description Language

Based on a discussion with this project's supervisor at the DLR, regarding the current structure of the DDL, the decision was made to completely redesign and reimplement it. The current DDL is application-specific and the redesigned and reimplemented is PUS-conform.

5.4.1 Application-specific DDL

The application-specific DDL targets specific applications in the OBSW. The code generator is adapted precisely to the requirements of the application. This allows a more sophisticated generation of code of the model.

The scoping of this DDL is similar to the one required in PUS. But it also allows the definition of TC & TM packets and parameters on all levels of the model's AST, which is in contrast to PUS. The defined TC & TM packets can be referenced in a service and the parameters in a TC or TM packet. Not referencing them leads to dangling definitions. This structure gives a lot of freedom to the domain expert but complicates the usage of this DDL.

Another difference to the PUS is that referencing of applications is not possible, because there is no element that can hold these references. Sometimes it is necessary to use two instances of the same application. For example a camera application, when two equivalent cameras are integrated in the satellite.

Furthermore, it is not possible to assign the addressing of the applications, services and sub-services in the format of $uint8_t$ IDs in addition to the names of the definition. But this is important for the transmission and the on-board routing of the packets. Also the syntax is not matching the PUS.

Additionally, features like the declaration of ranges and enums for the parameters are integrated in this DDL. They are used to find the correct data

type for the parameter in the code generator. But in the case of the test-framework, the instantiation of the parameters takes place in the test model. In case a value of a parameter instantiation is not within the range or the enum of the system model, a validator has to inform the domain expert about this issue. Generally, this would be a good idea, but the validator is defined as out of the scope for the project.

5.4.2 PUS-conform DDL

The mentioned problems of the application-specific DDL reveal already a lot of the design of the the PUS-conform DDL. For the rest of this report, DDL refers to the PUS-conform DDL.

The redesigned DDL is following a top-down approach, mirroring the routing structure, and therefore the CDH subsystem, of the TM & TC packets in the OBSW. The top node of the AST holds the references to the applications. The applications hold the references to the services. And the services contain the definitions of the TM & TC packets. This is the hierarchical structure displayed in the Background Chapter in Figure 2.1.

TM & TC packets can have an arbitrary amount of parameters, depending on the needs of the domain expert's component. The TC & TM packets can be uniquely identified according to their IDs: APID (application), SERID (service) and SUBSERID (TC or TM packet itself). The sequence count is added by the test-framework automatically to identify the oracle packet against which the arriving TM packet has to be compared. The syntax is adapted to the PUS, offering a recognition value, which is helping the domain expert to use the DDL.

The output is the AST of the system model, for the DTL to traverse. Additionally, the TC and TM packets are generated as non-initialized classes, which are used for the reinitialization of arriving TC or TM packets at the OBSW or the TEEP. Lastly, the actual structure of the behavioural part of the OBSW is generated with the CDH subsystem to rout the TC and TM packets through the OBSW.

5.5 Data Testing Language

The DTL consists of three sub-grammars: the *Data-centric*, the *Modified Linear Temporal Logic (MLTL)* and the *Relational Expressions* sub-grammar. Figure 5.3 reveals the structure of the DTL graphically.

The Data-centric sub-grammar is instantiating the in the DDL model defined packets. It cannot be used to define packets itself, it is only able to reference existing ones of the system model, by the investigation of the DDL model instance's AST. For the instantiation of the parameters, the Data-centric sub-grammar requires help by the Relational Expressions sub-grammar. Within the Relational Expressions sub-grammar, values and value ranges can be assigned to the parameters. When all parameters of a referenced TC or TM packet have values or a value range, they become TTC and Oracle packets. The packets are ready to be used in the MLTL sub-grammar. In the MLTL sub-grammar, the actual test cases can be defined, which are basically concatenations of TTC packets for the OBSW and expected Oracle packets in return from the OBSW. The MLTL sub-grammar is able to describe the expected temporal behaviour of the OBSW in a formal notation. The output of the DTL are compilable test cases in C++ and a test suite coverage report for the domain expert.

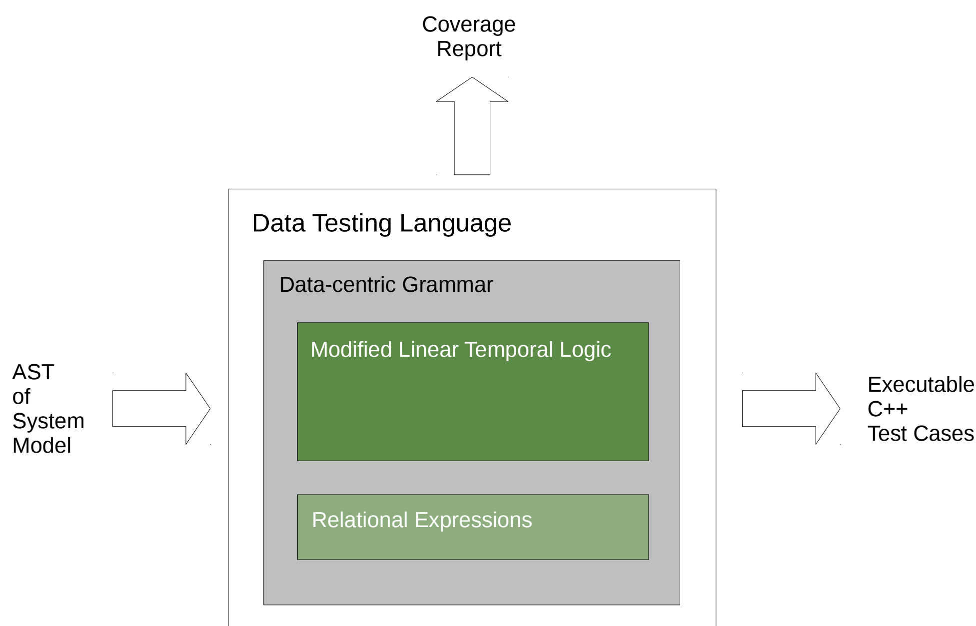

Fig. 5.3 Overview of the Data Testing Language

5.5.1 Data-centric Sub-grammar

The Data-centric sub-grammar is used, as aforementioned, to instantiate the defined TC/TM packets, to obtain TTC and Oracle packets. To enable that, the AST of the system model has to be cross-referenced. Here, cross-referencing

means that the results of the parser rules of one grammar/model are reachable by the other.

To instantiate a specific TTC/Oracle packet, the specific TC/TM packet needs to be identified and referenced. Identification and referencing is determined by the scoping of a grammar. Therefore, the layout of the referencing mechanism in the Data-centric sub-grammar is adapted to the scoping of the DDL.

The grouping of TTC/Oracle packets is the scoping of the Data-centric sub-grammar. Instantiated packets have to be named. A TTC/Oracle packet must be in a group, to enforce the oversight of the test model. To reference an instantiated packet in a test case, a fully qualified name is requested, which consists of the packet name and the group name.

The initialization of the parameters of the TTC/Oracle packets is closely related to the definition of the parameters. This facilitates the usage for the domain expert by the exploitation of repetitive patterns and contributes to the fulfilment of the requirements of a steep learning curve and the "easy-to-use" paradigm. To assign the concrete value or value range to the parameter, the Relational Expressions sub-grammar is used.

5.5.2 Relational Expressions Sub-grammar

Parameters have to be initializable with a direct value, an open range or a closed range. Therefore, the Relational Expressions sub-grammar is subdivided in two groups, unary and binary expressions.

A unary expression consist of one operand and one operator. The selectable operand can be a simple integer or a real value (also negative values are possible). The operator can be selected of a list of operators containing: "equal", "unequal", "less than", "less equal", "greater than" and "greater equal". The entire list is only reachable for the Oracle packet parameters, TTC packet parameters can just select the "equal" operator. This is due to the fact that TTC packets have to be precise for the OBSW. With unary expressions, the initialization with direct values and open ranges is covered.

A binary expression consists of two operands and two operators. The selectable operands are the same as for unary expressions and therefore simple integer and real values (also here, negative values are possible). The selectable operators are two unary operators, following the theory of sets in math. The first operand can be the "greater than" and the "greater equal" operator. The second operand can be the "less than" and the "less equal" operator. This is creating ranges and allows the initialization with the desired closed ranges.

5.5.3 Modified Linear Temporal Logic Sub-grammar

The MLTL sub-grammar is used to describe the behaviour of the OBSW. And more precise the properties that have to hold at a specific point in time, respectively at a specific state of the OBSW. It uses the prior defined TTC and Oracle packets as atomic proposition input for it's formula. LTL and CTL got investigated as a formal basis for the test-framework. Because of the specification-based and fault-based test case selection criteria, it is very difficult to exploit the quantifiers of CTL. To be able to profit of the additional option to declare properties that do not have to hold on all paths but at least at one, all potential paths have to be investigated. This makes CTL suitable for approaches with model checkers or SAT solvers, which are used with different coverage and test selection criteria as selected in this project. Therefore, not CTL but LTL serves as a basis for the test-framework. However, the MLTL is not as free as the LTL, it can be seen as a restricted subset of it. This means that not all operators are accessible for the test designer and the ones which are accessible can only be used in certain contexts.

OBSW as Kripke Structure

To understand the linking between the MLTL and the OBSW, the OBSW can be described using the formal notation of Kripke structures. It has been already denoted that the SUT is the OBSW in the user mode. The goal is to test the functionality and the structure of the SUT. Several other components in the user mode are additionally necessary to actually operate a satellite. In the subsequent Test Execution & Evaluation Platform section, some of these components are explained, representing a sufficient subset of all additional components to simulate the expected behaviour of an OBSW.

Figure 5.4 displays the behavioural part of the OBSW in the formal notation of a Kripke structure. The set of atomic propositions is a three tuple $APs = \langle \alpha, \beta, \tau \rangle$. α is a set of all evaluated addressing parameters of packets (APID, SERID, SUBSERID and Sequence Count). This set can be further subdivided in α_{OR} for the Oracle packets, and α_{TTC} for the TTC packets. β is a set of all evaluated payload parameters of the data field of the packets. This set can be further subdivided in β_{OR} for the Oracle packets and β_{TTC} for the TTC packets. τ is a set of all evaluated temporal parameters for periodic TM packets. The set of states is a three tuple $S = \langle S_{TTC}, S_{OR}, S_0 \rangle$. S_{TTC} is a set of all TCC states. S_{OR} is a set of all Oracle states. The states are visualized from the perspective of the test-framework, therefore they are not TC and TM states, but internally they are. The TCC states can not be investigated by the test-framework. They

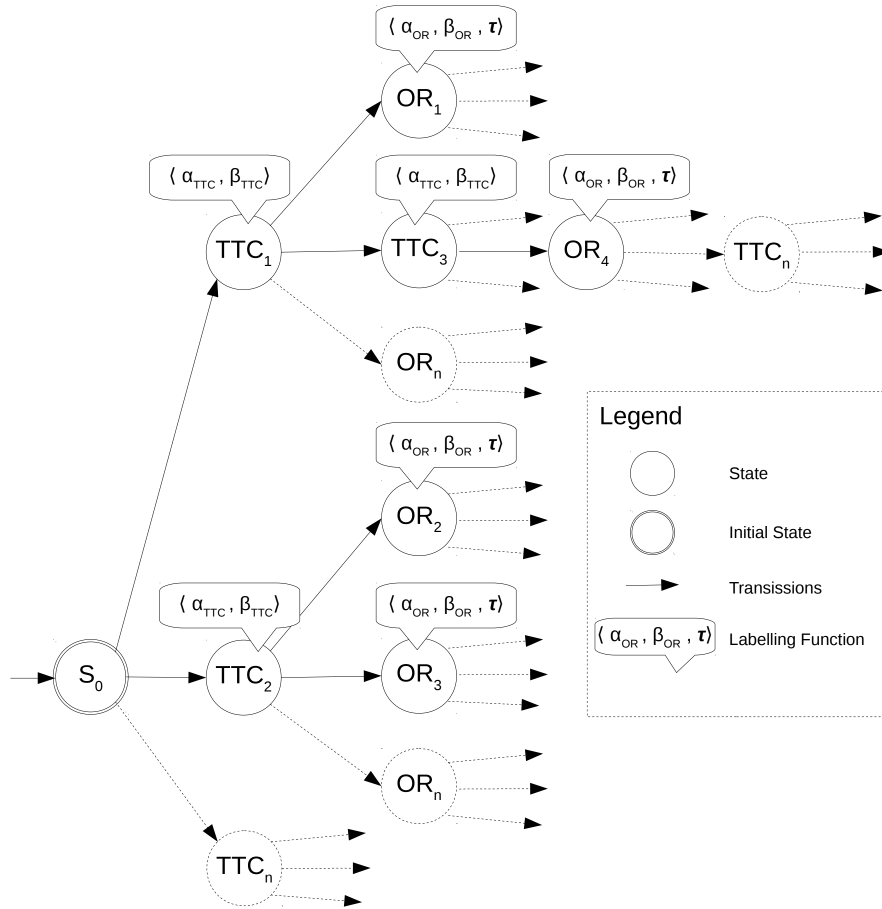

Fig. 5.4 Behavioural part of the OBSW, represented as Kripke structure.

are expected to hold as long as the Oracle states hold. S_0 is the initial state and is unknown. Other internal states of the OBSW are also unknown for the test-framework. The set of labelling functions F maps a subsets of APs to the different states. The transition relation T connects the states with each other.

Syntax and Semantics of MLTL

In this subsection the syntax and the semantics of the MLTL is explained. The used temporal and logical operators are:

Operator type	Syntax
Temporal	$\bigcirc$ ("next") ϕ
Temporal	$\Box$ ("always") ϕ
Temporal	$\Diamond$ ("eventually") ϕ
Temporal	ϕ U ("until") ψ

Logical	$\phi \wedge \psi$
Logical	$\neg\phi$
Logical	$\phi \rightarrow \psi$

Table 5.1 Temporal and logical operators in MLTL

The subsequent inference rules determine the expressiveness of the MLTL sub-grammar. In contrast to the basic LTL calculus, the MLTL needs further restrictions in their inference rules. This is because the domain experts need guidance and therefore limitations in the modelling process to facilitate the usage of the test-framework.

1. If the atomic proposition sets α_{OR}, α_{TTC}, β_{OR}, β_{TTC} and τ are each element of $APs = \langle \alpha, \beta, \tau \rangle$, then α_{OR} is a formula $\phi_{\alpha OR}$, α_{TTC} is a formula $\phi_{\alpha TTC}$, β_{OR} is a formula $\phi_{\beta OR}$, β_{TTC} is a formula $\phi_{\beta TTC}$ and τ is a formula ϕ_{τ}.
2. If $\phi_{\alpha OR}$, $\phi_{\beta OR}$, $\phi_{\alpha TTC}$ and $\phi_{\beta TTC}$ are formula, then $\phi_{\alpha OR} \wedge \phi_{\beta OR}$ is a formula ϕ_{OR} and $\phi_{\alpha TCC} \wedge \phi_{\beta TTC}$ is a formula ϕ_{TTC}.
3. If ϕ_{TTC} and ϕ_{OR} are formula, then $\phi_{TTC} \rightarrow \Diamond\phi_{OR}$ is also a formula Φ.
4. If ϕ_{OR1} and ϕ_{OR2} are formula, then $\phi_{OR1} \Diamond \phi_{OR2}$ is also a formula ϕ_{OR}.
5. If ϕ_{TTC1} and ϕ_{TTC2} are formula, then $\phi_{TTC1} \bigcirc \phi_{TTC2}$ is also a formula ϕ_{TTC}.
6. If Φ_1 and Φ_2 are formula, then $\Phi_1 \bigcirc \Phi_2$ is also a formula Φ.
7. If ϕ_{OR} and ϕ_{τ} are formula, then $\Box(\phi_{OR} \wedge \phi_{\tau})$ is a formula Ψ.
8. If $\phi_{\alpha OR}$ and $\phi_{\beta OR}$ are formula, then $\Box(\phi_{\alpha OR} \wedge \neg\phi_{\beta OR})$ is also a formula Ψ.
9. If $\phi_{\alpha OR}$ and $\phi_{\beta OR}$ are formula, then $\Box(\phi_{\alpha OR} \wedge \phi_{\beta OR})$ is a formula Ψ.
10. If Ψ and ϕ_{OR} are formula, then $\Psi U \phi_{OR}$ is a formula Γ.

Specific Behavioural Expressiveness Requirements

To explain the application of the aforementioned inference rules of the MLTL, they are set into context with the requirements of the Analysis and Requirements Chapter (4). This ought to increase the understandability of the inference rules.

TM Verification requirement: This is the basic requirement and follows the property of liveness. One TTC is transmitted and one specific TM packet expected in return. If the formula holds, the test case is passed. It is covered by (1.), (2.) and (3.).

TC/TM Sequence requirement: It must be possible to send multiple TTC packets and receive multiple TM packets in return. The test-framework is only in control of the TC side, meaning it can trigger the transmission of TTC packets. Therefore the $\bigcirc$ ("next") operator can be placed between TTC packets to model a sequence of TTC packets. The OBSW side is not in control of the test-framework, therefore it is not possible to predict the exact state change within the OBSW. To overcome this problem a less strict operator as $\bigcirc$ ("next") is needed. The solution is the $\Diamond$ ("eventually") operator between the Oracle packets. The used inference rules for this case are (1.), (2.), (3.), (4.) and (5.).

Procedures requirement: A procedure determines the point in time for the test-framework when to react on the arrival of one specific TM packet and can be seen as wait on statement for the test-framework. The reaction is a TTC packet transmission. The result of the functional evaluation is independent of the transmission, as it just has to arrive. This wait on statement is realized with the $\bigcirc$ ("next") operator between formula. This can by achieved by the application of (1.), (2.), (3.), (4.), (5.) and (6.).

Periodic TM Packets requirement: The periodically arriving TM packets (housekeeping packets) must be coverable by the test-framework. The domain expert can define an Oracle state that has to be reached within a certain time. The expected time cycle is attached to the Oracle packet parameters as an additional temporal parameter τ. The inference rules for that are (1.) and (7.).

The **Missing TM packets** requirement is handled by the TEEP and is not integrated in the MLTL sub-grammar.

Additional Temporal Requirements

Never requirement: The test designer has to be able to define a TM packet that should never arrive. This is following the LTL property for safety, meaning that something bad never happens. It is covered by (1.) and (8.).

Always requirement: The test designer is able to define a TM packet that should always arrive like defined in an Oracle packet. This can be achieved similar as the never requirement, whereas the $\neg$ operator is removed. It is covered by (1.) and (9.).

Until requirement: The test designer must be able to define a Oracle state that is reachable until another Oracle state is reached. This requirement is an

additional feature for the Never, Always and Periodic TM Packet requirements. It is integrated in the grammar with (1.), (7.), (8.), (9.) and (10.).

5.5.4 Coverage Report

The coverage report is created by a comparison of the defined TC & TM packets in the system model, and the instantiated and in test cases used TTC & Oracle packets of the test model. The domain expert is able to overview the coverage by a percentage value. In case a TC or a TM packet is not covered by a test case, it is listed in the report and marked as not covered. The total coverage per TC or TM packet is visualized by an additional number literal per packet, determining the amount of test cases that use an instance of that packet.

5.6 Test Execution & Evaluation Platform

The TEEP simulates the operation of a satellite and represents therefore the ground station. Transmission delay and atmospheric disturbance aspects do not have to be covered, as they are simulated by an extra hardware device. The ground station transmits TC packets over the base station to the satellite's telecommunication system, which forwards it to the CDH subsystem to route it to its final on-board destination. The satellite might then return a TM packet to the ground station following the same path, but backwards. The ground station evaluates the arriving TM packets and reacts accordingly to them. In case of the TEEP, the outcome of the evaluation is based on the test case scenario and the oracle, and happens fully automatic.

Figure 5.5 displays an overview of the components of the TEEP and the OBSW. Subsequent, the different components of the TEEP and the OBSW are elaborated regarding their functionality. Some of the components are equivalent in the OBSW and the TEEP, they are denoted separately as shared components. The OBSW is explained here as it is designed by the author following guidelines of working colleagues and it is important for the implementation.

5.6.1 Shared Components

TC & TM Blueprints

The TC & TM blueprints are necessary for the Packet Handler to re-instantiate the serialized packets which arrive in the TEEP or the OBSW. They are com-

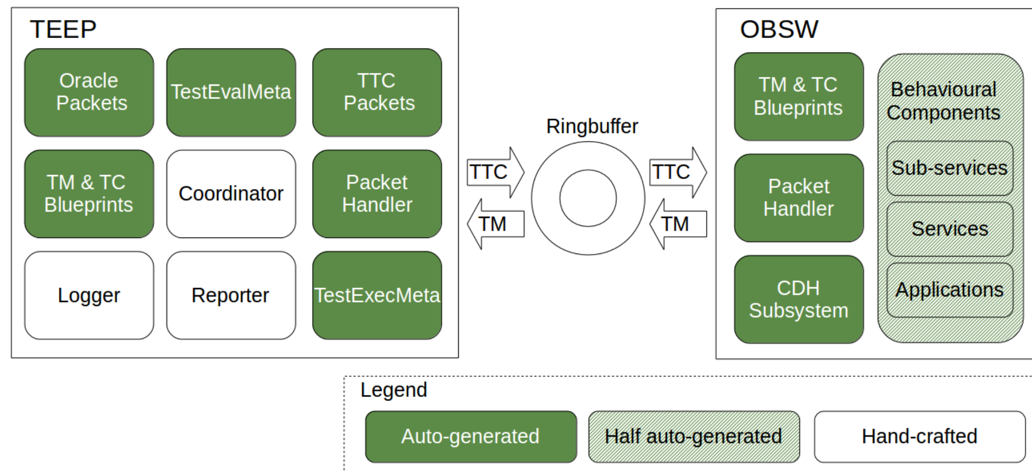

Fig. 5.5 Overview of the Test Execution & Evaluation Platform

pletely auto-generated by the test-framework, respectively by the code generator of the DDL.

Packet Handler

The Packet Handler is part of the communication system of the test-framework. It accesses the Ringbuffer to read and write the serialized packets from and to the buffer. The serialization mechanism of the packets is included in the TC & TM blueprints. The Packet Handler for the OBSW and the TEEP are both auto-generated by the code generator of the DDL.

Ringbuffer

The Ringbuffer is used as a shared memory for the communication between the OBSW and the TEEP. It is the only interface between the SUT and the TEEP. The entire transmission of packets is logged in a specific XML log. The Ringbuffer component is hand-crafted.

5.6.2 OBSW Components

CDH Subsystem

The CDH subsystem receives the instantiated TC packets from the Packet Handler and routes them to the addressed Application, Service and Sub-service of the Behavioural component of the OBSW. It is also called by the applications to revert that process, to send TM packets in return to the TEEP. This component is auto-generated by the DDL's code generator.

Behavioural Component

The Behavioural component represents the actual SUT. The structure in the design is following the PUS. It is hierarchically structured in Applications, Services and Sub-services. In case of an arriving TC packet, the payload of the TC packet changes the state of this part of the OBSW. The structure of this component is generated automatically by the code generator of the DDL. This allows the programmer to implement the functionality, with no need to care about the interfaces for the packet routing any more.

5.6.3 TEEP Components

TTC Packets

The TTC packets are the test case inputs. They are auto-generated by the Relational Expressions sub-grammer of the DTL. These TTC packets have the same structure as the TC Blueprints, but they are initialized with the by the domain expert defined values.

Oracle Packets

The Oracle packets are the expected output of the test cases. They contain the expected data and share structural parts of the TM packets. Their purpose is to enable the evaluation of the arriving TM packets from the OBSW, which are re-instantiated of the byte-stream of the ringbuffer with the help of the TM blueprints. The Oracle packets are auto-generated by the Relational Expressions sub-grammar.

TestExecMeta

The TestExecMeta holds the information for the test case execution. It is auto-generated from the test case definition of the MLTL sub-grammar. The TestExecMeta knows when to send a TTC packet and when not to. This decision is based on a state change within the OBSW, about which the TEEP gets informed via TM packets.

TestEvalMeta

The TestEvalMeta is the counter part of the TestExecMeta. It holds the necessary information for the evaluation of the arriving TM packets against

specific Oracle packets. It is completely auto-generated by the MLTL sub-grammar.

Coordinator

The Coordinator is the worker of the TEEP. It spins on the TestExecMeta and the TestEvalMeta. Its initial task is to send all currently by the TestExecMeta allowed TTC packets via the Packet Handler to the SUT. These transmissions are logged. When this task is finished, it waits for a response of the OBSW. When the response arrives, it is evaluated by the help of the TestEvalMeta and the Oracle packets. The result of the evaluation is reported. When all parameters of all arrived packets are evaluated and reported, the process starts from the beginning. This continues until there are no more Oracle packets left that expect an evaluation or a watchdog timer ends the test run.

Logger

The logger notes the destination/source address and a TEEP time-stamp of all arriving and leaving packets. Two logs are created by the Logger, the expected TC and TM packet transmissions and unexpected TM packets from the OBSW, which are not covered by the test cases. Those packets can not be instantiated with the TM blueprints, but identified by the investigation of the byte stream on their APID, SERID and SUBSERID.

Reporter

The Reporter is called by the Oracle Packets and creates a report with the result of the test evaluation. The provided data in a report entry is the TM packet source address, the Oracle Packet name, the parameter name, the actual value, the expected value and the line number of the test case in the test model. As the test evaluation is online, it also prints the coloured test result immediately to standard out. This is important for the tester, to see as early as possible whether a test failed, leaving the choice to him or her to cancel the entire test run.

Chapter 6

Implementation

In this chapter the implementation of the test-framework is elaborated. Beginning with the DDL, then the DTL and its sub-grammars and finally the TEEP.

6.1 Used Tools & Implementation Workflow

In the Background Chapter, the used tools to implement the design are presented (2.6). For the implementation of the grammars and the code generators, the Xtext framework is used. The grammar implementation takes place in the Xtext grammar editor and is in EBNF. The explicit scoping and the code generator is written with the Xtend language, as it offers strong concepts like type inference and mighty template expressions. The TEEP is written in C++ and implemented in Microsoft Visual Studio.

The implementation workflow of the test-framework is subdivided into three phases. Each phase's output is the input for the next phase. The three phases mirror the components of the test-framework: DDL phase, DTL phase and TEEP phase.

After the first run through all phases, the prototype of the test-framework was finished. The completion of the second run lead to the final version of the test-framework. The subsequent explanation of the implementation follows the order of the three phases and explains the final version of the test-framework.

6.2 Data-Description Language

The implementation of the DDL focuses on the grammar of the DDL and the code generators. An explicit scoping and semantic validator are not necessary in the DDL for the test-framework.

6.2.1 DDL Grammar

The DDL grammar definition takes place in the Xtext grammar editor in EBNF form. The finished grammar is transformed into an Ecore meta-meta-model. This meta-meta-model is used to create the DDL editor in which the domain expert defines the system model. The finished system model is transformed into an Ecore meta-model. This meta-model is then accessible via the cross-referencing by the later described DTL.

To oversee the DDL grammar, Figure 6.1 displays the AST with the parser rules. The grammar itself is, in contrast to the DTL grammar, not following any specific calculus. It reflects the hierarchical structure of the PUS, and enables the definition of TC & TM packets.

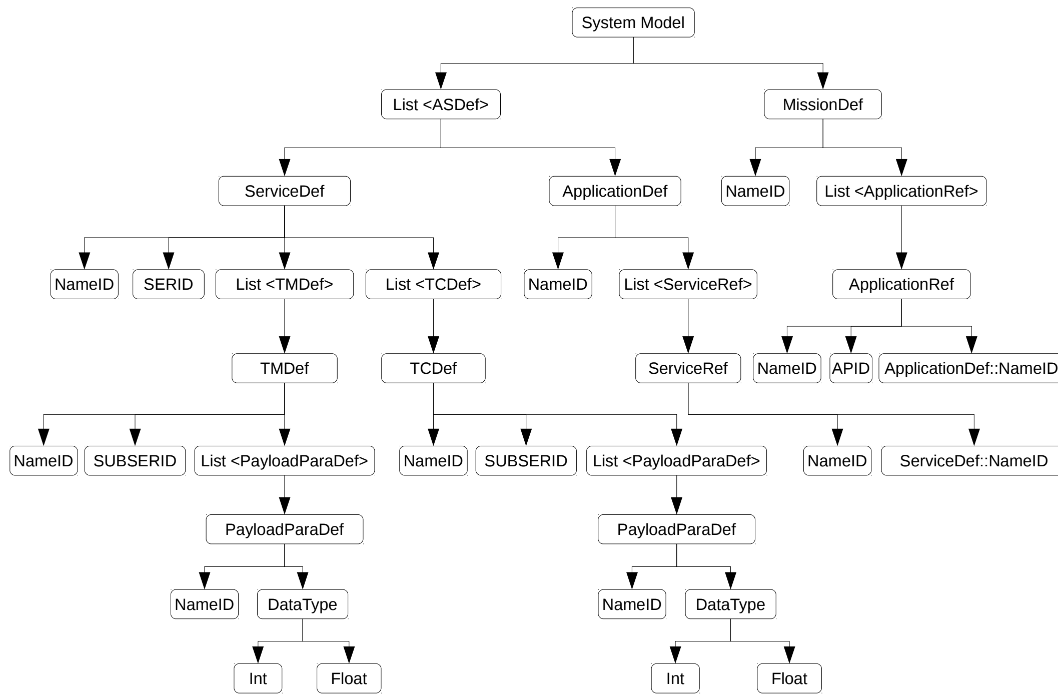

Fig. 6.1 Abstract syntax tree of DDL.

A *System Model* consists of one *MissionDef* and a list of *ASDefs*. A *MissionDef* has a name to identify it and a list of *ApplicationRefs*. An *ApplicationRef* refers to an *ApplicationDef* and has to be named and additionally identified with an uint8_t value, which is the APID. The referencing mechanisms of existing *ApplicationDef* is realized with implicit scoping, which is automatically generated by the Xtext framework in the background. This implicit scoping is then accessible within the grammar via cross-references to any other parser rule of the AST.

An *ASDef* can either be an *ApplicationDef* or a *ServiceDef*. An *ApplicationDef* has a name and a list of *ServiceRefs*. A *ServiceRef* refers to a *ServiceDef* and has

to be named too. Also here, the referencing mechanism is realized with implicit scoping.

A *ServiceDef* consists of a name, an uint8_t SERID and lists of *TCDefs* and *TMDefs*. *TCDefs* and *TMDefs* are similar. The defined packet requires a name and an uint8_t SUBSERID for the identification. Additionally, *PayloadParaDefs* can be defined, which are collected in a list. A *PayloadParaDef* has to be named and a data-type assigned, which can be int or float.

The names of all definitions in the DDL and the DTL model are just for the domain expert to keep the overview. The actual test cases use the APID, SERID, SUBSERID and Sequence Count for the identification of the packets. To visualize the result of the applied DDL, Figure 6.2 shows a system model with one application, two services and their packets.

```
 1 // mission definition
 2 mission is
 3
 4     applicationreference laser as Laser with apid 10
 5
 6 end
 7
 8 // application defintion
 9 application Laser is
10
11     servicereference laserControl as LaserCtrlSer
12     servicereference housekeepingRef as HousekeepingSer
13
14 end
15
16 // service defintion
17 service LaserCtrlSer with serid 20 is
18
19     // telecommand packet definition
20     telecommandpacket TCLaserCtrl with subserid 30 is
21         parameter setState is int
22     end
23
24     // telemetry packet defintion
25     telemetrypacket TMLaserStatus with subserid 40 is
26         parameter state is int
27         parameter Temp is float
28         parameter PeakPower is float
29     end
30 end
31
32 service HousekeepingSer with serid 21 is
33
34     // telemetry packet definition
35     telemetrypacket LaserMeanvalues with subserid 41 is
36         parameter MeanTemp is float
37         parameter MeanPeakPower is float
38     end
39 end
40
```

Fig. 6.2 System model example.

6.2.2 Code Generator

The code generation is triggered when the system model has been saved and the static syntax validator is not throwing any errors. The generated files can be found in the *src_gen* folder of the project of the system and test model.

The code generator of the DDL generates the Packet Handler with the Packet Factory of the TEEP and the OBSW, used to re-instantiate a packet as class objects of the serialized byte stream of the Ringbuffer. The Packet Handler waits on thread events when either the OBSW or the TEEP transmitted a packet over the Ringbuffer.

Additionally, the CDH subsystem of the OBSW is generated. The CDH resolves the packet address, APID, SERID and SUBSERID, to route it to the destination in the behavioural component of the OBSW. Therefore, it is distributed as handler methods on each level of the behavioural component.

The behavioural component itself is created as hierarchical structure. The applications are classes that hold instances of service classes. The service classes have methods as members, representing the sub-services. The programmer is able to modify and implement the actual behaviour of the OBSW in this part of the OBSW.

Finally, the by the packet factory used TC and TM packets to instantiate the arriving TC and TM packets of the OBSW/TEEP are generated. The payload parameters are members of the classes. For each payload parameter, related getter and setter methods are generated as additional members. The addressing parameters reside in an *AbstractPacket* class of which the TC and TM packets inherit. This is also the case for addressing parameter related getter and setter methods. The TC and TM packet classes have an empty constructor with no parameters for it. Additionally, the virtual methods $serialize()$ and $deserialize()$, which also reside in the *AbstractPacket* Class, are generated as members. These methods are needed to serialize and de-serialize the packet objects to create a byte stream for the transmission from the TEEP to the OBSW and vice versa. To enable that, each implemented virtual method stores all parameters in a uint8_t array. The unknown-size problem of the payload parameters is resolved with an index for the array and the $sizeof()$ method. The actual storing in the array is conducted using $memcopy()$.

After the code generator is finished with all components, the DDL copies them to a specific location in the TEEP.

6.3 Data-Testing Language

The implementation of the DTL focuses on the grammar of the DTL, the explicit scoping and the code generation. Not every of these three aspects is needed in every sub-grammar. The DTL grammar, and therefore the three sub-grammars, is also defined in EBNF form. The output of the grammar definition is similar to the DDL grammar and the AST in the from of a Ecore meta-meta-model, used to create the DTL editor. This DTL editor can then be used by the domain expert to create the test model. The DTL takes the AST of the DDL system model and makes it accessible in the DTL editor via an import statement and Xtext framework settings (grammar cross-referencing). The imported AST is in case of the DTL accessible via the *DDL_X* variable. To increase the understandability of the following subsections, explaining the sub-grammars, an example of a test model in Figure 6.2 shall be given in advance.

6.3.1 Data-centric Sub-grammar

The implementation of the Data-centric sub-grammar consists of a grammar definition and explicit scoping.

Sub-grammar

Figure 6.4 displays the AST of the sub-grammar. The *Test Model* holds a list of groups. A group can either be a *TTCGroup*, an *OracleGroup* or a *TestGroup* declaration. These groups increase the overview of the test model.

A *TTCGroup* has to be named and owns a list of *TTCDefs*. An *OracleGroup* has to be named too and owns a list of *OracleDefs*. A *TTCDef* must be named and the *TTCParameters* defined. As already mentioned, the DTL is not able to create new packets, but can reference existing ones of the system model. This is achieved by referencing an *ApplicationRef*, a *ServiceRef* and a *TCDef* of the system model. This is similar to an *OracleDef* definition, but the *OracleDef* is not referencing a *TCDef* but a *TMDef*. According to the referenced *TCDef* or *TMDef*, a list of either *TTCPayloadParas* or *OraclePayloadParas* is available. These *PayloadParaDefs* of the system model can then be instantiated with values or value ranges. At this point the Data-centric sub-grammar is left for the Relational Expressions sub-grammar. The in the Data-centric sub-grammar used referencing mechanism to instantiate a desired TC or TM packet is explained in the following Explicit Scoping subsection.

```
1 // Group of TTC packets
2 TTCGroup ttcGr1 is
3
4     // laser control
5     TTC swLaserOn to [laser.laserControl.TCLaserCtrl] is
6         setState set 1
7     end
8     // laser control
9     TTC swLaserOff to [laser.laserControl.TCLaserCtrl] is
10        setState set 0
11    end
12 end
13
14 // Group of Oracle packets
15 OracleGroup orGr1 is
16
17    // laser control
18    Oracle laserStateOn from [laser.laserControl.TMLaserStatus] is
19        state expect == 1
20        Temp expect [80 .. 100]
21        PeakPower expect < 10
22    end
23    // laser control
24    Oracle laserStateOff from [laser.laserControl.TMLaserStatus] is
25        state expect == 0
26    end
27    // laser housekeeping
28    Oracle cyclicLaserMeans  from [laser.housekeepingRef.LaserMeanvalues] is
29        MeanTemp expect == 90.563
30    end
31 end
32
33 // Group of test suites
34 TestSuitsGroup tsGR1 is
35
36    Testsuit ts1 is
37        // 1. TM verification
38        ttcGr1.swLaserOn implies orGr1.laserStateOn must hold
39        // 2. TC/TM sequences
40        ttcGr1.swLaserOn implies  orGr1.laserStateOn future orGr1.laserStateOff must hold
41        // 3. Procedures
42        ttcGr1.swLaserOn implies orGr1.laserStateOn must hold
43        next
44        ttcGr1.swLaserOff implies orGr1.laserStateOff must hold
45        // 4. Periodic TM packets
46        cyclic set milSec: 2000 orGr1.cyclicLaserMeans must hold
47        // 5. Always & Never
48        always orGr1.laserStateOn must hold
49        never orGr1.laserStateOff must hold
50        // 6. Always until
51        always orGr1.cyclicLaserMeans until orGr1.laserStateOff must hold
52    end
53 end
54
```

Fig. 6.3 Test model example.

The third group is the *TestGroup* and consists of a name and a list of TestSuiteDefs. A *TestSuiteDef* has a name and a list of *TestLTLExps*. When the test designer reaches a definition of a *TestLTLExp*, he or she is leaving the Data-centric sub-grammar for the MLTL sub-grammar.

Explicit Scoping

The referencing of the correct TM and TC packets of the cross-referenced system model is conducted, using explicit scoping. Scoping is generally achieved

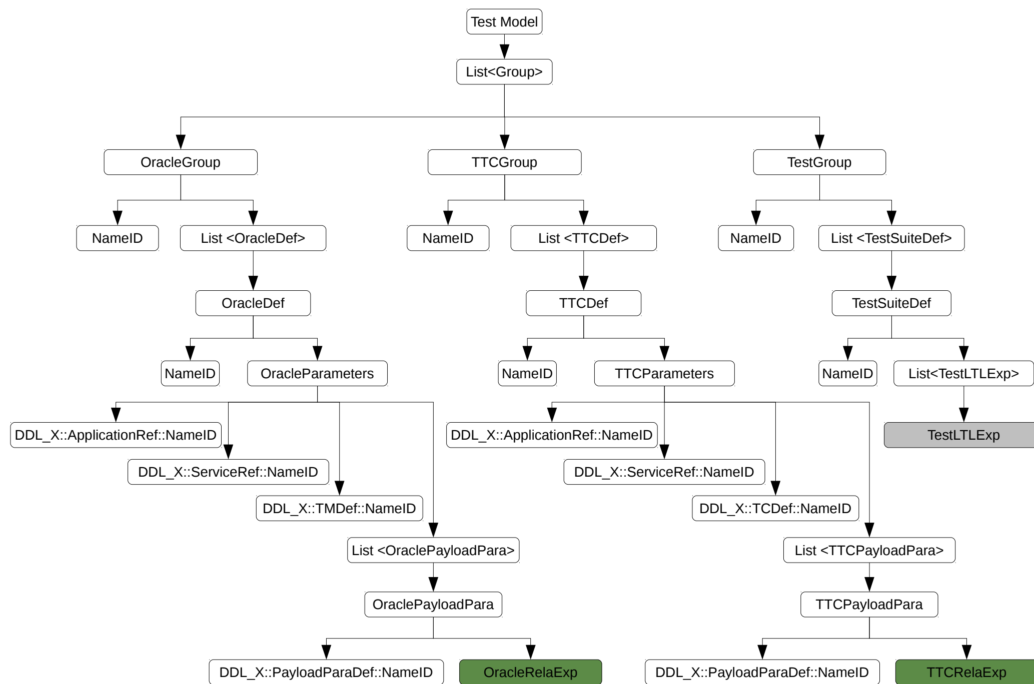

Fig. 6.4 Abstract syntax tree of the Data-Centric sub-grammar.

by a scoping method which gets called if a parser rule wants a specific object, created by another parser rule, of the AST. The scoping method can traverse the AST from the position of the specific object downwards. Each parser rule of a Xtext grammar gets a scoping method by default, created by a code generator in the background. This is implicit scoping. The method can be called by a cross-reference definition in the grammar of the form:

$$result = [<RefParsRule>]$$

When the model is instantiated, the selected *<RefParsRule>* object is stored in the *result* variable. In the case of the test-framework, the DDL grammar is cross-referenced with the DTL, but the scoping methods of the DDL are not available in the DTL editor. Therefore, an explicit scoping method needs to be defined for the Data-centric sub-grammar. To call the explicit scoping method, a naming convention is integrated in Xtext. If the cross-referenced grammar's AST is imported and the root node of the AST is stored in the *DDL_X* variable, then the call in the grammar looks like this:

$$<ParsRule> : result = [DDL_X:<RefParsRule>];$$

Referencing a parser rule which has to be on the top-level of the cross-referenced AST. If such a call is integrated in a grammar, Xtext searches for the scoping method:

def Iscope scope_<ParsRule>_result(<ParsRule> object, EReference ast)

This is the reason why there are different parser rule names (*TTCParameters* and *OracleParameters*) for the addressing of the packets. In the scoping method for the TTC packets the TM packets are filtered out. And in the scoping method of the Oracle packets, the TC packets are filtered out. When the desired TC or TM packet is referenced, a list of the selectable *PayloadParaDefs* appears.

Additionally to mention is, that the *ApplicationRefs* are not on the top-level of the cross-referenced AST. Therefore, all explicit scoping methods have to traverse the AST to the *MissionDef* to reach the there-stored *ApplicationRefs* as a starting point of the scoping.

6.3.2 Relational Expressions Sub-grammar

The implementation of the Relational Expressions sub-grammar consists of a grammar definition and a code generator.

Sub-grammar

The sub-grammar has the task to enable the assignment of values or value ranges to the payload parameters of the TC and TM packets of the DDL system model, which are referenced in the Data-centric sub-grammar. When this is done, the TTC and Oracle packets are finished and can be used by the code generator to transform them in C++ classes.

Figure 6.5 displays the AST of the Relational Expressions sub-grammar. In case of a *TTCRelaExp*, the only option is to assign a *Numberliteral*, which can be either an int or a float value. This is due to the fact that the TTC packets have to have a precise value as input for the OBSW. The *IntValue* and *Realvalue* parser rules of the *Numberliterals* are integrated with a cross-referencing of the AST of the basic Ecore model. There, the parser rules for integer and real values are already defined (*EFloat* and *EInt*). Whether the selected value fits the format of the defined TC packet is not validated, as a validator is out of the scope of this project, what is the same for the *OracleRelaExp*.

In the *OracleRelaExp* either a *UnaryExp* or a *BinaryExp* can be selected. In case of a *unaryExp* a *unaryExpOperator* has to be selected and a *Numberliteral*. If a *binaryExpr* has been selected a *LBracketOperator*, followed by two *Numberliterals* and a *RBracketOperator* is necessary.

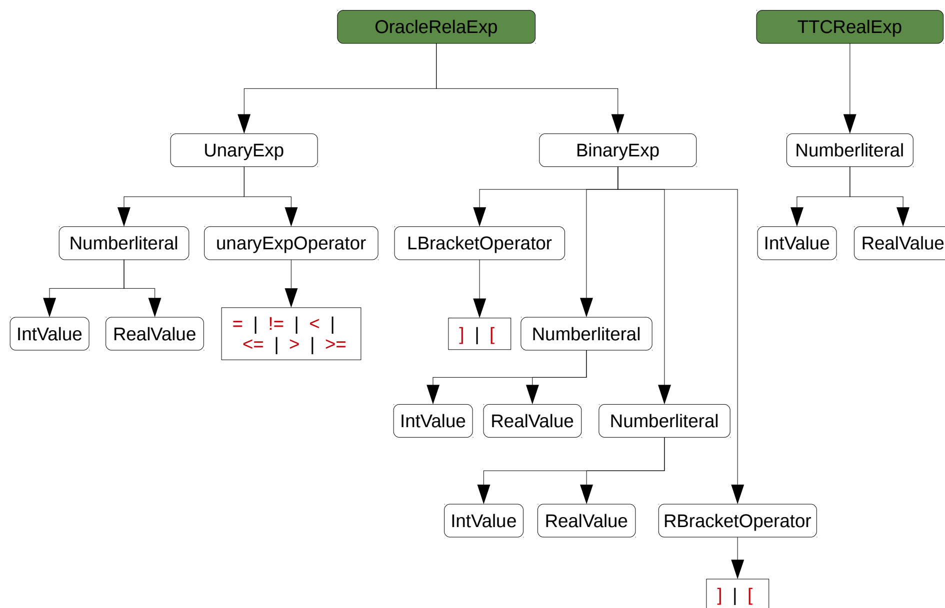

Fig. 6.5 Abstract syntax tree of the relational expressions.

Code Generator

The code generation of the Relational Expressions sub-grammar is triggered when the test model has been saved and the static syntax validator is not throwing any errors. The generated files can also be found in the *src_gen* folder of the project of the system and test model.

The code generator creates the TTC and Oracle packet classes. To generate the TTC packet classes, the TC packet classes are enriched by an additional constructor which takes all parameters, the one of the *Packet Header* and of the *Packet Data Field*, as arguments. These are the instantiated addressing and payload parameters. In the body of the constructor are the correlating setter methods to instantiate the object with all parameter values. These specific constructors are called by a constructor of the TTCAll class, that holds all TTC packet objects as members.

The Oracle packet classes have only methods to compare all parameters of an arriving TM packet with the expected values as members. The return value of these methods is a truth value, which is an AP for the MLTL sub-grammar. The comparison is based on if-conditions with the selected operators of the Oracle packet parameter instantiation. Also here, the Oracle packet classes inherits the *Header Field*-related parameter comparison methods of a specific abstract class. Their constructors are also empty, which is obvious as they only have methods as members. Problematic is that the TEEP does not know how many payload parameters are in one packet, therefore it lacks the information

on which compare methods to call. To overcome this issue, the specific abstract class contains also a virtual compare all method, which is implemented in each Oracle class. The implemented compare all method has all comparison methods of the packet and the specific abstract packet in its body. The TEEP only calls the compare all method, in which also the Reporter is triggered after each parameter evaluation.

The last step for the code generator of the Relational Expressions sub-grammar is to copy the created classes at a predefined position in the TEEP, making them ready for usage in the TEEP.

6.3.3 Modified Linear Temporal Logic Sub-grammar

The MLTL sub-grammar implementation consists of a grammar and a code generator.

Sub-grammar

Figure 6.6 displays the AST of the MLTL in the same notation as the ASTs of the Data-centric and Relational Expression sub-grammar. It is expected that the reader is now able to use the gained knowledge on how to read such an AST from the prior sub-grammar subsections. Therefore, this AST is not explained in detail.

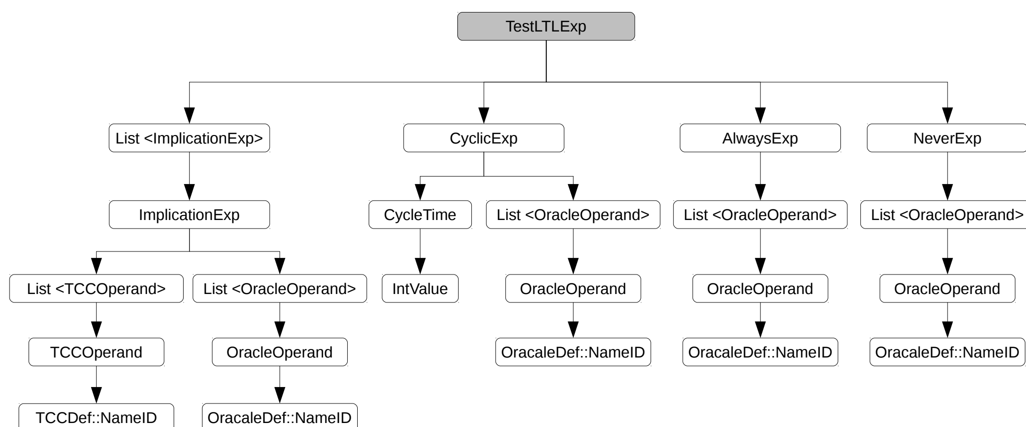

Fig. 6.6 Abstract syntax tree of the Modified Linear Temporal Logic.

Here, the focus lies on the explanation of the transformation of the theoretical design of the MLTL in concrete syntax of the grammar. To enable that, the concrete syntax is set in context to its theoretical MLTL formal notation and the requirement. The subsequent formal notation is already a simplification,

as the scoping of the APs is done by the data-centric sub-grammar. The output of the Data-centric sub-grammar is: ϕ_α, representing a group of all addressing parameter APs of one packet; ϕ_β, representing a group of all payload parameter APs of one packet;

To illustrate the **TM Verification requirement**, it is in formal notation:

$$(\phi_{\alpha TCC} \wedge \phi_{\beta TTC}) \rightarrow \Diamond(\phi_{\alpha OR} \wedge \phi_{\beta OR})$$

The equivalent expression is in concrete MLTL syntax:

```
<TTCPk> implies <ORPk> must hold
```

The **TC/TM Sequence requirement** is elaborated with a sequence of two TTC packets and two Oracle packets as example. But the sequences can also be arbitrary long. This is in formal notation:

$$((\phi_{\alpha TCC1} \wedge \phi_{\beta TTC1}) \bigcirc (\phi_{\alpha TCC2} \wedge \phi_{\beta TTC2})) \rightarrow \Diamond(\phi_{\alpha OR1} \wedge \phi_{\beta OR1})\Diamond(\phi_{\alpha OR2} \wedge \phi_{\beta OR2})$$

The equivalent expression is in concrete MLTL syntax:

```
<TTCPk1> next <TTCPk2> implies <ORPk1> future <ORPk2> must hold
```

To show the **Procedures requirement** in formal notation, two TM verifications are concatenated. But the concatenation can be arbitrary long and also consist of TC/TM sequences.

$$(\phi_{\alpha TCC1} \wedge \phi_{\beta TTC1}) \rightarrow \Diamond((\phi_{\alpha OR1} \wedge \phi_{\beta OR1}) \bigcirc ((\phi_{\alpha TCC2} \wedge \phi_{\beta TTC2}) \rightarrow \Diamond(\phi_{\alpha OR2} \wedge \phi_{\beta OR2})))$$

The equivalent expression is in concrete MLTL syntax:

```
<TTCPk1> implies <ORPk1> next <TTCPk2> implies <ORPk2> must hold
```

The **Periodic TM Packets requirement** is in formal notation:

$$\Box(\tau \wedge \phi_{\alpha OR} \wedge \phi_{\beta OR})$$

The equivalent expression is in concrete MLTL syntax:

```
cyclic set milsec:<int> <ORPk> must hold
```

The **Never requirement** is in formal notation:

$$\Box(\phi_{\alpha OR} \wedge (\neg\phi_{\beta OR}))$$

The equivalent expression is in concrete MLTL syntax:

```
never <ORPk> must hold
```

The **Always requirement** is in formal notation:

$$\Box(\phi_{\alpha OR} \wedge \phi_{\beta OR})$$

The equivalent expression is in concrete MLTL syntax:

```
always <ORPk> must hold
```

The **Until requirement** is applicable for the three requirements, Never (1), Always (2) and Periodic TM Packets (3). All three are in formal notation:

$$\Box((\phi_{\alpha OR1} \wedge \neg\phi_{\beta OR1})U(\phi_{\alpha OR2} \wedge \phi_{\beta OR2})) \quad (1)$$
$$\Box((\phi_{\alpha OR1} \wedge \phi_{\beta OR1})U(\phi_{\alpha OR2} \wedge \phi_{\beta OR2})) \quad (2)$$
$$\Box((\tau \wedge \phi_{\alpha OR1} \wedge \phi_{\beta OR1})U(\phi_{\alpha OR2} \wedge \phi_{\beta OR2})) \quad (3)$$

The equivalent expressions for these three use-cases are in concrete MLTL syntax:

```
never <ORPk1> until <ORPk2> must hold (1)
always <ORPk1> until <ORPk2> must hold (2)
cyclic set milsec:<int> <ORPk1> until <ORPk2> must hold (3)
```

Code generator

The code generator of the MLTL sub-grammar transforms the test case formulae into two C++ classes. The *TestEvalMeta* and the *TestExecMeta* class. These classes contain a list of structs with specific members. The classes member methods can search the lists top-down and update specific members of the structs. Each list entry corresponds to one implication. A test case can contain multiple implications. To explain the transformation of the abstract test cases into compilable C++ code, each classes struct and the enclosed members get explained.

The evaluation struct in the *TestEvalMeta* class can be investigated in Table 6.1. It is searched and updated, depending on the arriving TM packets from the OBSW.

Member	Explanation

`struct <TMPkgAddress>{`
`uint8_t apid;`
`uint8_t serid;`
`uint8_t subserid;`
`}tmPkgAddress`
	The <TMPkgAddress> struct is used to identify the Oracle packet based on the APID, SERID, SUBSERID and Sequence Count of the arriving TM packet. The list is searched for the matching address.

<TMPkgAbstract>* orPkg	This pointer points to the Oracle packet object which is needed for the evaluation. It is of the type of the inherited abstract class, because the concrete Oracle packet objects are of different classes.
<bool> active	The active flag states a list entry to be accessible for evaluation. It is by default "true" and is set to "false" if the Oracle packet is not allowed to be reached any more. This is the case, if the second TM packet of an until expression arrived.
<int> rmvActiveLstNum	The rmvActiveLstNum is only important for the second packet of an until expression, in other cases it is set to zero. It carries the index of the list entry which ones active flag has to be set to "false", because this is the first packet in the until expression and not allowed to be reached any more.
<double> timeDeadline	The timeDeadline is the τ value and therefore the defined interval time in which the cyclic TM packet has to arrive.
<double> timeStamp	The timeStamp is updated by the test-framework at the beginning of the test run. It is used for the evaluation of the timeDeadline and is always refreshed when a cyclic TM packet arrives.

`<bool> arrivedOnce`	The arrivedOnce flag helps to identify missing packets and is by default "false". It is set to "true" if the packet arrived at least once in the test run. All packets that remain in state "false" are printed in the reportNeverArrived.xml report. While the test run, this report is shrinking by every arrived packet.
`<bool> neverExp`	The neverExp flag determines a test case is created by the usage of the NeverExp. In case it is "true", the evaluation of the TM packets against the Oracle packets is negated. All other Expressions of the MLTL have this flag set to "false".
`<bool> TTCLess`	The TTCLess flag is set to "true" for the NeverExp, the AlwaysExp and the CyclicExp. These expressions do not need a prior TTC packet transmission to trigger a specific TM packet in return. The reason for a TM packet transmission can be for example a technical on-board event of the satellite. If this flag is set to "true", the Sequence Count is ignored for the evaluation of the packet, only the APID, SERID and SUBSERID have to fit.

`<std::string> testSuiteGroup`
`<std::string> testSuite`
`<std::string> testCaseModelLine`
	These strings are important for the back-tracing of an evaluation entry in the reports to the test model. They contain the names of the test suite group and the specific test suite as well as the line number of the test case in the test suite of the test model.

Table 6.1 Evaluation struct of the *TestEvalMeta* class.

The execution struct in the *TestExecMeta* class can be investigated in Table 6.2. It is searched in the beginning of the test loop to trigger the transmission of TTC packets which are in TTCstate "ready". It is also updated on arriving TM packets to change the dependabilities and to modify the TTCstate.

Member	Explanation
`<int> TTCState`	Based on the value of the TTCState, the TEEP triggers the transmission of a TTC packet or not. The different value of the TTCstate have following meanings: 0 = "not ready", 1 = "ready", 2 = "waiting on TM response", 3 = "processed".
`<TCPkgAbstract>* tcPkgAbst`	This pointer points to the TTC packet object which has to be transmitted if the TTCstate == 1. It is of the type of the inherited abstract class, because the concrete TTC packet objects are of different classes.
`std::list <TMPkgAdr> TMWaitList`	The TMWaitList is a list of all TM packets on which the TTC packet has to wait for arrival before it can be transmitted. This is needed for the procedures. In case a TM packet of the list arrives, it is removed from it. When the list is empty the TTCState is set from "not ready" to "ready".
`std::list <TMPkgAdr> TMEvalList`	The TMEvalList is needed to determine whether all expected packets, triggered by a TTC packet, arrived. In case the list is empty, the TTCState is set from "waiting on TM response" to "processed".

<uint8_t> seqCount	The Sequence Count of the TTC is stored separately from the actual TTC packet in this execution list. The packets are identified regarding the APID, SERID, SUBERID and Sequence Count in the evaluation. To enable a reusage of TTC packets with different expected TM packets in different test cases, the Sequence Count has to be changed from test case to test case. The value that is stored to identify the Oracle packet for the evaluation is the token offset in the test model of the test case expression. The programmer has to copy the Sequence Count of the TTC packet, which is overwritten by this seqCount short before the transmission of the TTC packet, into the TM packet Sequence Count in the behavioural OBSW.

Table 6.2 Evaluation struct of the *TestEvalMeta* class.

The final task for the code generator of the MLTL sub-grammar is to copy the generated C++ classes to a specific location in the TEEP.

6.3.4 DTL Structural Coverage Report

The structural coverage report is created by a comparison of the TM and TC packets definitions in the system model, with the in test cases used TTC and Oracle packet instantiations. The coverage criteria is fulfilled if all TM packets are referenced in at least one Oracle packet instance, which is then used in a test case. Additionally, the TC packets have to be referenced in at least one TTC packet instance, which is referenced in a test case.

To enable that, the DDL creates a list of all TC and TM packets of the system model via a small code generator. For each found packet, the application, service and sub-service names are collected. In the test model, the test cases

are traversed and the there referenced TTC and Oracle packets investigated. The names of the system packets, which are referenced in the TTC and Oracle packets are compared to the aforementioned list. In case a packet of the system model is not covered by at least one test case, it is specifically marked in the coverage report. All other system model packets are denoted with the amount of test cases that target them.

This determines the structural coverage of the system model by the test model. The concrete output is the Coverage Report for the domain expert. It consists of a percentage coverage value, a list of not referenced system model packets and a list of all TM and TC packets with the amount of test cases they are referenced in. In case the percentage coverage value is less than 100%, the list of not referenced system model packets reveals the uncovered packets. Figure 6.7 shows a screen-shot of the coverage report.

```
 1
 2 ***********************************************************************
 3 *               The Test Coverage                                     *
 4 ***********************************************************************
 5
 6                      - 100% -
 7
 8 ***********************************************************************
 9 *               Not covered Telecommand Packets                       *
10 ***********************************************************************
11
12                      - none -
13
14 ***********************************************************************
15 *               Not covered Telemetry Packets                         *
16 ***********************************************************************
17
18                      - none -
19
20 ***********************************************************************
21 *           Test case coverage per Telecommand Packet                 *
22 ***********************************************************************
23     4 laser laserControl TCLaserCtrl
24
25 ***********************************************************************
26 *           Test case coverage per Telemetry Packet                   *
27 ***********************************************************************
28     8 laser laserControl TMLaserStatus
29     2 laser housekeepingRef LaserMeanvalues
30
31
```

Fig. 6.7 Coverage report for the domain expert.

6.4 Test Execution & Evaluation Platform Implementation

The general design of the TEEP and the OBSW can be found in the Design Chapter (5.6). The focus of this section is on the implementation of this design. Some components are equal in the OBSW and in the TEEP, therefore they only get explained once.

6.4.1 Shared Components

TC & TM Packet Blueprints

These groups of classes contain the TC and TM packets, used for the re-instantiation by the packet factory of the packet handler. They are used in the TEEP as well as in the OBSW. These classes have been elaborated in detail in the code generator subsection of the DDL (6.2.2).

Ringbuffer

The ringbuffer is a shared component, used for the communication between the OBSW and the TEEP. The TEEP has one instance of it to send its TTC packets to the OBSW. And the OBSW has one instance to transmit the TM packets in return. In case the TEEP wants to send a TTC packet to the OBSW, the serialized byte stream is copied to the ringbuffer, the position of the end of the packet updated and the OBSW informed about the transmission of the TTC packet via a thread event. The OBSW reads the packet and updates the front position of the ringbuffer, respectively freeing the memory block. In case the end position equals the front position, both positions are set to zero (empty ringbuffer). The OBSW and the TEEP loop on their ringbuffers until they are empty. When they are empty the thread event is set to zero. Problematic is when the TEEP writes to the ringbuffer, updating the end position and setting the event. And the OBSW thread gets the CPU back after it just has read the ringbuffer, which it expects to be empty now, and resets the thread event. The TTC packet in the ringbuffer will not be read by the OBSW until another TTC packet transmission sets the thread event again. This might hang the test run and is a classical race condition. Therefore, the ringbuffer is protected by a mutex, including the setting and resetting of the events, to make it thread-safe. The size of the ringbuffer can be defined by the programmer via a #define. It should be sufficiently large to hold multiple (100 for example) of the maximum

allowed packet size. This is important as the OBSW runs single threaded and if the execution of one TTC takes very long the TEEP throws packets away when the ringbuffer is full.

CDH Subsystem

The CDH subsystem mirrors the structure of the behavioural part of the OBSW. It is used for two tasks. The first is to identify an arriving packet. This identification is important for the packet factory of the Packet Handler, to re-instantiate the serialized byte stream. The second task is to route the packet trough the behavioural OBSW, to its final destination. The second task is not required by the TEEP, only by the OBSW. Therefore, it is not available for the TEEP. But one has to admit, that when the CDH subsystem already found the packet, the routing task is also complete. This is true. But in a real OBSW, the CDH subsystem also stores the arriving packets and transmits them in the form of micro threads to the behavioural part of the OBSW (tasking framework). This is done to schedule the arriving packets based on their priority level (channelization). To enable that, the behavioural OBSW has virtual handler methods on the application and service level, which resolve the packet address again. This routing mechanism is also realized in the OBSW of this project, but the creation of micro threads is not.

Packet Handler

The packet handler has access to the ringbuffer and re-instantiates the arriving TC and TM packets via its packet factory. The packet factory is using the CDH subsystem on the OBSW as well as on the TEEP side to find the correct TM or TC packet blueprint class. In case it found a matching class, it instantiates an empty object of it and fills it with the help of the packet's de-serialize method. In case it can not find a matching class, it throws it away on the OBSW side and logs it on the TEEP side. The packet handler has the method waitOnEvent, in which the TEEP and the OBSW thread stall until a ringbuffer event has been triggered.

6.4.2 OBSW Components

Behavioural OBSW

The behavioural OBSW consist of application classes and service classes with additional sub-service methods as members, mirroring the PUS structure

of OBSW. The application objects have variables and methods as members. They also hold a method to route the packet to a specific service object. This method is a part of the CDH subsystem, as already denoted. The same counts for the service objects, which route the packets to their sub-service methods. In the sub-service methods, the payload content is evaluated. Based on the evaluation, the service manipulates the state of the application. It is possible to manipulate the application object, because a pointer to the object has been routed too. This manipulation might trigger the creation of a TM packet in the application. The application is able to do that by the call of a sub-service which is part of a service it holds a reference of. Then, the sub-service calls the packet handler to transmit a specific TM packet to the TEEP.

6.4.3 TEEP Components

Oracle and TTC Packets

The Oracle and TTC packets are classes, which are created by the code generator of the Relational Expressions sub-grammar (6.3.2). They have been explained in that section in detail.

TestEvalMeta and TestExecMeta

The TestEvalMeta class and the TestExecMeta class are created by the code generator of the MLTL sub-grammar (6.3.3). They have been explained in that section in detail.

Logger and Reporter Classes

These classes are in charge for the generation of the logs and the reports. The logger produces two XML files, one with all arriving and resolvable TM packets and all departing TTC packets. The other file consists of arriving TM packets that could not be instantiated, because the CDH subsystem could not resolve the addressing. Figure 6.8 shows a log of all arriving and departing packets as example.

The Reporter creates three files, one with all results of the evaluations of arriving TM packets against their designated Oracle packets. Another with all evaluation results that failed, to enable a fast access to the failed test cases for the programmer. And the third with all TM packets that have been expected to arrive but never did. All three files are in XML format. Additionally, all test

```xml
<?xml version="1.0" encoding="UTF-8"?>
<testsuites name="tsGR1">
        <testsuite name="ts4">
                <testcase oraclename="cyclicLaserMeans" telemetryddress="10.21.41"
                        testcaseLine="56" timestamp="Thu Jul 02 13:43:27 2015"
                        testresult="passed" parametername="Interval time"
                        expectedvalue="2000" actualvalue="1025"/>
                <testcase oraclename="cyclicLaserMeans" telemetryddress="10.21.41"
                        testcaseLine="56" timestamp="Thu Jul 02 13:43:27 2015"
                        testresult="passed" parametername="Apid"
                        expectedvalue="10" actualvalue="10"/>
                <testcase oraclename="cyclicLaserMeans" telemetryddress="10.21.41"
                        testcaseLine="56" timestamp="Thu Jul 02 13:43:27 2015"
                        testresult="passed" parametername="Serid"
                        expectedvalue="21" actualvalue="21"/>
                <testcase oraclename="cyclicLaserMeans" telemetryddress="10.21.41"
                        testcaseLine="56" timestamp="Thu Jul 02 13:43:27 2015"
                        testresult="passed" parametername="Subserid"
                        expectedvalue="41" actualvalue="41"/>
                <testcase oraclename="cyclicLaserMeans" telemetryddress="10.21.41"
                        testcaseLine="56" timestamp="Thu Jul 02 13:43:27 2015"
                        testresult="passed" parametername="MeanTemp"
                        expectedvalue="== 90.0000" actualvalue="90.0001"/>
        </testsuite>
</testsuites>
```

Fig. 6.8 Reporter output of reportAll.xml.

results are forwarded to the standard output. Figure 6.9 shows the report with all evaluations of one cyclic packet, including the interval time.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<testrun>
        <packetevent type="telemetry" timestamp="Thu Jul 02 13:43:27 2015"
                address="10.21.41">
        <packetevent type="telecommand" timestamp="Thu Jul 02 13:43:28 2015"
                address="10.21.32">
</testrun>
```

Fig. 6.9 Logger output of logAll.xml.

Google Test and other unit test frameworks got investigated to incorporate them in the test-framework. The investigation revealed that the testing process in such frameworks does not meet the requirement of an online evaluation of the test cases. In case of Google Test, the test case's evaluation is triggered by a final *RUNALLTESTS* macro. This affords the storage of all arriving TM packets until the test execution is completely finished. An online investigation of the results is then not possible, what means that the programmer does not receive any information about the test run while it is executed. It would be possible to create the unit test environment for every parameter evaluation, outputting the result and destroying the environment again. But the overhead in the integration of the API of the unit test-framework in such a way is higher than the actual benefit of it. The benefit is just a standardized XML output. Therefore, the layout of the Google Test output is used as a template and a custom made Reporter and Logger is implemented.

Coordinator

The Coordinator class is the heart and the worker of the TEEP. Its only member is the runTest method, in which almost the entire test process on the TEEP side is implemented. It initializes all components of the test-framework and controls the transmission and the evaluation of the packets in a loop. In the subsequent section, this process in the loop is explained.

6.4.4 Test Process

Figure 6.10 shows the test process in great detail. The loop is implemented in the Coordinator's runTest method. The prior mentioned components are integrated in that loop to fulfil their task in the TEEP.

When the test run begins, the OBSW, ringbuffer, Logger, Reporter, TTC packets and Oracle packets are instantiated. The OBSW is spawned in an extra thread. In case periodic TM packets got defined as test cases for the test run, a watchdog timer thread is instantiated with twice the maximum interval time, comparing all interval times of the periodic TM packets. Then the thread is spawned and all periodic TM packet time stamps in the TestEvalMeta list get updated.

In case no periodic TM packets got defined in any test cases, the watchdog timer is initialized with a default value. The default value and the multiple of the maximum interval time is configurable. The watchdog timer thread is spawned and the noCyclicTMFlag set.

At this point the actual loop in the Coordinator begins. The TestExecMeta is searched for TTC packets which are in the TCCState "ready". They get transmitted via the Packet Handler and the Ringbuffer and logged via the Logger. This continues until all TTC packets in TTCState "ready" got transmitted. After the transmission of a TTC packet, the TTCState is set to "waiting on TM". There is no possibility to define a test case with a TTC packet that is not expecting a TM in return. This is due to the fact that the domain expert shall be forced to write a test with measurable output of the reactive system.

If no TTC packet is waiting for a TM packet to arrive and the noCyclicTMFlag is set, the test run is finished, as nothing is expected to happen any more. In case this condition is false, the TEEP begins to wait for an event. In case the watchdog timer throws an event, the test run is finished. The programmer has to be aware that the watchdog time might be too short for some TM verifications. Therefore, he or she has to make sure that the multiple of the maximum interval

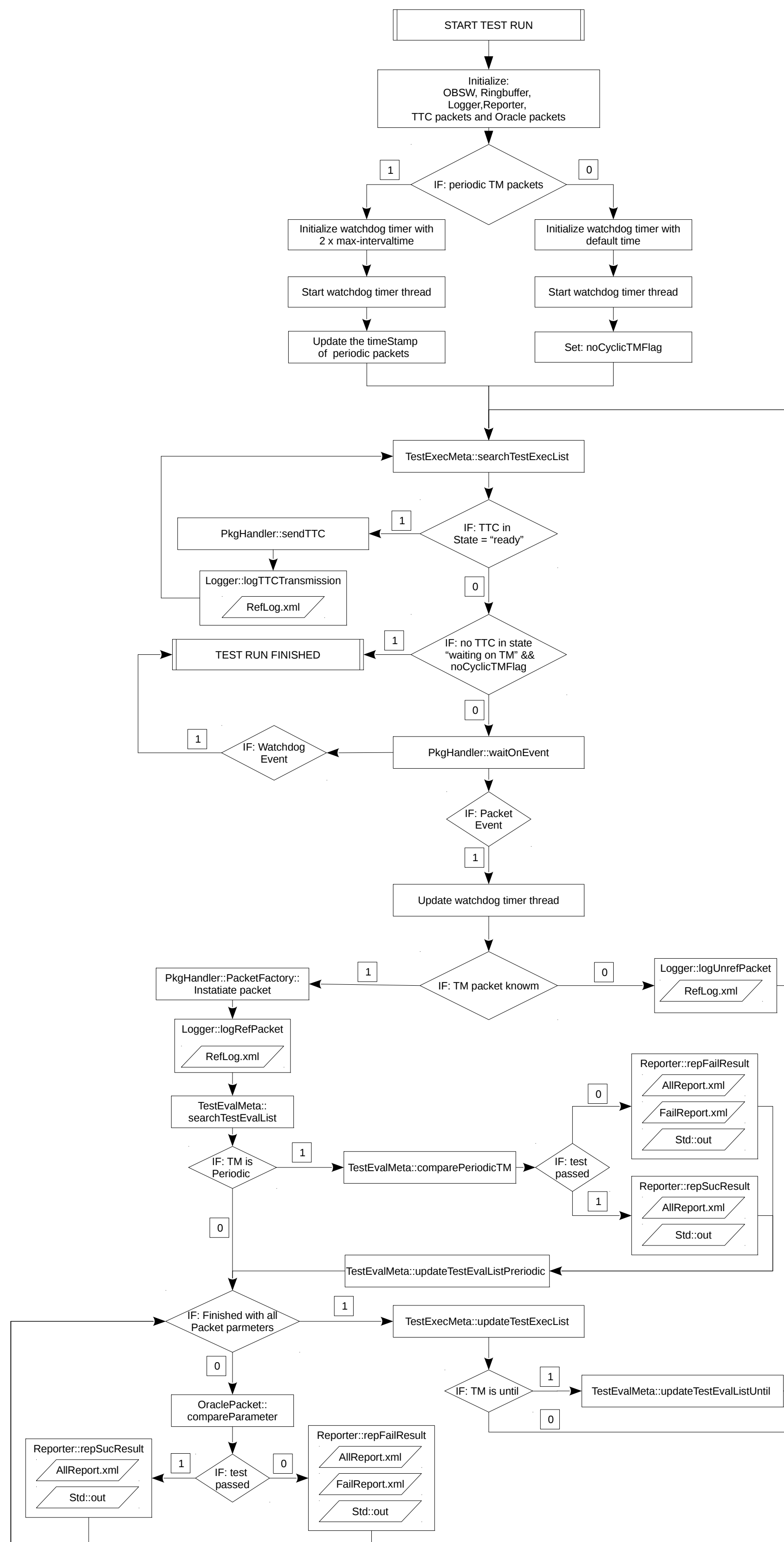

Fig. 6.10 Flowchart of testing process.

time is sufficient. This is a drawback of the watchdog timer, but without it, the programmer would have to guess the time and abort manually.

In case of an arriving TM packet event, the watchdog timer is updated to its initial value and the byte stream investigated to determine the TM packet. In case it is unknown, the byte stream is logged via the Logger in the UnrefLog.xml file and the program jumps to the beginning of the loop. In case it is known, it is instantiated by the packet factory and the de-serialize method and logged in the RefLog.xml.

If the TM packet is referencing to a periodic Oracle packet, the time stamp is evaluated and a report entry written. Additionally, the time stamp in the TestEvalMeta list is updated. After this, the evaluation is for all arriving TM packets the same. The Oracle packet is used to evaluate the TM packet and calls the Reporter after each parameter evaluation. The reporter either writes to std:out and the AllReport.xml or to the std:out, AllReport.xml and FailReport.xml. This depends whether the evaluation creates a success or a failure verdict.

When all parameters have been evaluated the TestExecMeta has to be updated. This might change the TTCState of some TTC packets. In addition, the TestEvalMeta list is updated if the TM packet is the second operand of an until expression. Therefore, the prior explained active flag has to be set to false and the Oracle cannot be reached any more. At this point, the end of the loop is reached and the process begins again from the top.

Chapter 7

Evaluation

This Chapter targets the requirements coverage, the mastering of the challenges, identified in the Related Work Section of the Background Chapter, and the limitations of the test-framework.

7.1 Automatic Specification-based Challenges Evaluation

In the Related Work Section of the Background Chapter, the challenges for an automatic specification-based test-framework are elaborated (2.8). Based on these challenges the test-framework is theoretically evaluated, to reveal the coverage of them.

The challenge of the **specification language** is mastered by the application of a formal language as a basis. This allows a precise definition of test cases, overcoming ambiguities of informal languages.

The **method for generating adequate test sets** is mainly handed over to the domain expert, with the specification-based and fault-based test selection and termination criteria. But also structural test cases are created by the test-framework in the background. Additionally, the domain expert is helped to determine the coverage of the test suite by the Coverage Report.

The **translation between the abstract data and the concrete data** is handled by the transformation of the TTC and Oracle packets in compilable and therefore executable C++ classes. Also, the gap between the abstract test cases and the executable test cases is mastered, with the translation of the temporal behaviour of the test cases in C++ lists for the execution and the evaluation.

The **analysis of test results to determine bugs** requires the definition of an oracle as an expected output. The Coverage Report helps the domain expert

to determine whether all Oracle packets are created or which ones are still missing. But there can also be the situation that unexpected TM packets arrive, which are not covered by the test model. In case this happens, these packets are logged in a specific unresolvableTMLog.xml file. In this file, the entire *Header Field* and *Packet Data Field* of the arriving TM packet is denoted, enabling the programmer to trace the origin of the TM packet back to the application, service and sub-service. This log can then be used to discuss this event with the domain expert.

The **debugging** challenge is the tracing back of a failure to the model. The test-framework displays the model line of the test case in the test model as information for the programmer in every parameter evaluation.

The specific challenges for **integration testing** are hard to evaluate on a theoretical basis. With the used coverage criteria, the completeness of an integration test suite can not be guaranteed. In general, this challenge remains when using specification-based testing, due to the incomplete nature of such an approach. A concrete result for this aspect can be gained when the test-framework is used for the first time in an actual space mission.

7.2 Requirements Coverage Evaluation

The entire report emphasises on the requirements for the test-framework. In the Analysis and Requirements Chapter they are identified and applied in the Design Chapter. In the subsequent Implementation Chapter the design is linked to the actual implementation of the test-framework. In this section, the implementation is evaluated theoretically towards its requirements coverage.

7.2.1 General Requirements Coverage

The subsequent list denotes the evaluation of the general requirements regarding their fulfilment in the implementation.

- **Data-centric design:** The test-framework, and there more precise the DTL, follows the same data-centric approach as the DDL. The TTC and Oracle packets, as test case input and expected output, are in focus, clustered in groups. The entire test-framework emphasises on the transmission and routing of data packets from and to the OBSW.

- **As automatic as possible:** The test-framework is able to generate executable test cases by an automatic translation of the specification-based

definition of abstract test cases and the OBSW structure. The created files are automatically copied to their location in the TEEP, in which they can be executed immediately without any changes. The TEEP transmits the TTC packets to the empty behavioural OBSW structure, which returns an acknowledgement to the standard output from the destination of the TTC packet in the OBSW.

- **Enable test-driven development:** The test-driven development is strongly supported by the test-framework. It allows the creation of compilable test cases before one line of behavioural OBSW source code is written. The programmer has all use-case scenarios of the domain expert's component in the form of test cases available, before one line of source code of the actual OBSW implementation is written.

- **All outputs in XML:** All reports and logs are in XML and it is additionally possible to view them in HTML5 format, by setting of a #define in the TEEP.

- **Logs and reports:** The reports as well as the logs are generated automatically in an online fashion. This means that the reports are created after each iteration in the test loop of the Coordinator. This allows the programmer to abort a test run and still maintain all evaluation and transmission results, that took place until the stop of the test run. Additionally, the domain expert is provided with instant feedback on the results of the parameter evaluations in the form of coloured printouts to standard out.

- **Exploit re-usability:** There are two different forms of re-usability in the test-framework. On one side, the re-usability of structures within the models is targeted. The packets of the test model are created by an instantiation of existing TC and TM packets of the system model. Additionally, it is not necessary to create new TTC or Oracle packets for each test case. The packets can be reused arbitrary often in various test cases. On the other side, the re-usability of the system and test model of a mission for future missions is focused. It is possible to reuse test cases for frequently applied services in various applications, like the *Memory Management Service*. Only slight adaptations are necessary to integrate a service in another application. In the system model, the service needs to be referenced in the application definition. In the test model, the TTC and Oracle packets of the service have to be copied and the new application address adapted. In case different values or value ranges of the parameters are requested, the domain expert changes them. As the

behaviour of the service is the same in all applications, the test cases can also get copied and the new TTC and Oracle packets referenced. It is recommended to have a database of common services and their TTC packets, Oracle packets and test cases, to be able to just copy them into the models.

- **Functional testing:** The functional testing is in the foreground of the test-framework. The domain expert is able to specify a certain expected behaviour of the OBSW, reflecting the behaviour of his or her component.
- **Manual test creation:** Not all test cases are created manually. The structural tests, related to the on-board routing of the packets are automatically generated. But the functional tests have to be defined manually by the domain expert.
- **Domain expert as test designer:** The handling of the test-framework targets the domain expert as test designer, allowing him or her to generate complex test scenarios for his or her component in a facile way. The facility resides in the restrictions in the grammar. The structure in the DDL, to describe the system model, guides the domain expert through the process of creating OBSW, emphasising on its hierarchical structure. The generated C++ code is way more complex, but hidden before the domain expert's eye. The DTL is following the same restriction idea, grouping the test input, expected test output and the actual test cases. Additionally, the domain expert is able to name his or her application, services and sub-services without any convention. This helps to incorporate the domain knowledge into the model and facilitates also the understanding on the programmers side on the domain expert's component.
- **Simple to use tool:** The tool is integrated as an Eclipse IDE plug-in, which might be a drawback for this requirement. Such full-blown IDEs always comprise a certain complexity in the handling, and can quickly lead to a rejection by the operator, in case he or she is not used to them. But the actual necessary functionalities of the IDE for the generation of a project, respectively system and test model, are only a few. A project has to be generated and two files created, one with .ddl and the other with .dtl at the end. The actual generation of the models is then strongly supported by features like syntax highlighting, code folding, syntax validation and the display of options with the "Ctrl + Space" short cut in the editor. On the side of the programmer, a certain experience with IDEs can be assumed. The TEEP ought to be used in any IDE or also from the command line,

whereas Microsoft Visual Studio has been applied for the implementation. To launch a test run, the TEEP only needs to be compiled and executed. In case a specific test suite is of interest, other test suites can be commented out with the "not" keyword after a test suite definition in the test model.

- **Steep learning curve via repetition:** An emphasis of the test-framework are repetitive tasks while modelling. In case the basic principle, meaning the hierarchical structure of OBSW, is understood, the enrichment of the system model with additional applications, services and sub-services is straight forward. The gained knowledge of the generation of a system model can then be reused in the test model, especially in the instantiation of TTC and Oracle packets. They follow the basic design of a system packet, offering a strong recognition value for the domain expert. The definition of test cases is more difficult, because multiple options are possible. But keywords like "always", "never" and "cyclic" are self-explaining. Other keyword like "implies" or "must hold" provide information on the background of the grammar. Also the repetitive patterns for the programmer when actually executing a test run creates confidence in the programmer over time. But a drawback is the reading of the reports and logs. They are still self-explaining, but it has to be known why a packet is listed in this report or log and not in another. This is something that has to be learned by the programmer with the help of the manual.

7.2.2 PUS and Additional Temporal Requirements Coverage

The PUS requirements are specific for OBSW of satellites. The goal is to integrate these requirements in the test-framework, to allow the domain expert to specify different use-case scenarios of his or her component.

Packet Structure Requirement Coverage of PUS

The packet structure in the test-framework is strongly related to the subsequent elaborated behavioural requirements. This is due to the fact that some behavioural aspects of PUS have to be transmitted in a packet from the ground station to the OBSW in the satellite.

The Packet Field Header has a the *Type* as member which is important to determine whether a packet is a TC or a TM packet. In case of the implementation of the test-framework, this is of minor interest. As the packets that arrive at the TEEP from the OBSW are always TM packets and in the other direction always TC packets. But when programming a real OBSW, the CDH subsystem

sorts and intermediate stores the packets. Therefore, it is useful to implement it from the beginning.

The addressing parameters, *APID*, *SERID* and *SUBSERID* are all in the *Packet Header*, which is in contrast to the PUS protocol. But it is very common to do it like that, especially in smaller scientific space missions. The DLR conducts mostly those smaller space missions, with micro satellites as the maximum satellite size.

The *Sequence Count* is very important to determine whether a packet is lost or not. In the implementation of the test-framework it is used for that purpose, evaluating automatically if a packet is missing. Additionally, it is possible to detect if the OBSW is in an unexpected state, by the arrival of a not expected TM packet.

The *Payload Length* member defines the length of the *Packet Data Field* and helps to optimize the usage of the ringbuffer. Additionally, it can be easily used to determine the communication budget of the subsystems of the OBSW.

The *Packet Data Field* contains the payload parameters and is variable in size as requested by PUS. PUS defines 65KB as the maximum packet size, but it is in most missions much smaller. The programmer is able to control the maximum size of the *Packet Data Field* by the definition of a maximum packet size that can be written to the ringbuffer. In case the ringbuffer throws an error, the programmer has to use multiple packets to transmit the payload parameters.

Behavioural Expressiveness & Additional Temporal Requirements Coverage

The behavioural expressiveness requirements target specific modes and options of the PUS, offered to the ground station to control and supervise the satellite. Together with the additional temporal requirements, they represent the entire ability to express the behaviour of the reactive system over time. The subsequent list denotes the correlation between these requirements and the actual implementation. The enumeration in the list is mirrored in the test case definition of a test model in Figure 7.1. This test model is the same test model that has been presented in the Data Testing Language Section of the Implementation Chapter (6.3). To facilitate the reading, the test model is again displayed in this chapter.

1. **TM verification:** A TM verification triggers the transmission of a TTC packet and expects a specific TM packet in return, which is compared to an Oracle packet. In the test model, a TTC packet to switch the laser "on",

```
1 // Group of TTC packets
2 TTCGroup ttcGr1 is
3
4     // laser control
5     TTC swLaserOn to [laser.laserControl.TCLaserCtrl] is
6         setState set 1
7     end
8     // laser control
9     TTC swLaserOff to [laser.laserControl.TCLaserCtrl] is
10        setState set 0
11    end
12 end
13
14 // Group of Oracle packets
15 OracleGroup orGr1 is
16
17    // laser control
18    Oracle laserStateOn from [laser.laserControl.TMLaserStatus] is
19        state expect == 1
20        Temp expect [80 .. 100]
21        PeakPower expect < 10
22    end
23    // laser control
24    Oracle laserStateOff from [laser.laserControl.TMLaserStatus] is
25        state expect == 0
26    end
27    // laser housekeeping
28    Oracle cyclicLaserMeans  from [laser.housekeepingRef.LaserMeanvalues] is
29        MeanTemp expect == 90.563
30    end
31 end
32
33 // Group of test suites
34 TestSuitsGroup tsGR1 is
35
36    Testsuit ts1 is
37        // 1. TM verification
38        ttcGr1.swLaserOn implies orGr1.laserStateOn must hold
39        // 2. TC/TM sequences
40        ttcGr1.swLaserOn implies  orGr1.laserStateOn future orGr1.laserStateOff must hold
41        // 3. Procedures
42        ttcGr1.swLaserOn implies orGr1.laserStateOn must hold
43        next
44        ttcGr1.swLaserOff implies orGr1.laserStateOff must hold
45        // 4. Periodic TM packets
46        cyclic set milSec: 2000 orGr1.cyclicLaserMeans must hold
47        // 5. Always & Never
48        always orGr1.laserStateOn must hold
49        never orGr1.laserStateOff must hold
50        // 6. Always until
51        always orGr1.cyclicLaserMeans until orGr1.laserStateOff must hold
52    end
53 end
54
```

Fig. 7.1 Simple test model.

is transmitted and a TM packet, with the state and other parameters of the laser with specific values or value ranges, is expected to return.

2. **TC/TM sequences:** The TC/TM sequences are concatenations of TTC or/and Oracle packets. In the test model, the test case sends a TTC packet to the OBSW, to switch the laser "on". It expects two TM packets in return, one that states that the laser is actually "on" and another that denotes that the laser will be switched "off" at some point in the future.

The concatenations on both sides, the TTC and the Oracle side, can be arbitrary long.

3. **Procedures:** The Procedure in the test model is the most simple procedure possible, which are two TM verifications combined with the "next" operator. But it is also possible to append more TM verifications or TC/TM sequences. At the test model, in the first TM verification the laser is switched "on" and in the second switched "off" again.

4. **Periodic TM packets:** The periodic packets require the definition of the expected time interval in which the TM packet ought to arrive. There is no TTC packet needed. The result of the evaluation of all parameters is reported, including the interval time. This construct can also be seen as an always expression with an additional temporal requirement. The model shows an example with an interval time of 2000 milliseconds and an expected TM packet with certain mean values of the laser.

5. **Always & Never:** In the implementation the always and the never expressions have no TTC packet to trigger them. They are statements that hold from the beginning of the test run's execution until the end. The never expression negates the payload parameter definitions of the Oracle packet. It is also possible to define the Oracle packet with unwanted values or value ranges and to use the always expression. This would lead to the same results. In case of the example test model, it is expected that the laser should be always in the state "on" and never in the state "off".

6. **Until:** The until expression is an attachment of the always, never and periodic packets expressions, to determine a point in time (a state in the OBSW), at which the prior, in the entire test case defined state, should not be reached any more. In case the OBSW still transmits a TM packet that reveals that the OBSW reached that banned state, the arriving TM packet is handled as unknown and logged in the unresolvable TM packets log. The test model only displays the combination of always and until, but the other combinations follow the same rules. The test case asks for the laser mean values until the laser is switched "off".

7. **Missing TM packets:** The Missing packets are listed in the test-framework, which are all Oracle packets of the defined test cases at the beginning of the test run. The list is reduced by an entry if a specific packet actually arrives once. In case, an in a test case defined Oracle packet never arrived, it is in state missing and reported to the programmer. The identification

of the Oracle packets for the evaluation towards the arriving TM packets happens in the background by the framework with the help of the Sequence Count, which holds the offset of the test case expression in the test model as value. therefore it is not visible in the test model example.

7.3 Limits of the Test-framework

There are various limits in the test-framework and in the expressiveness of the created reports and logs.

One issue is that the temporal behaviour of the SUT is measured using the system time of the computer on which the test-framework is executed. Scheduling aspects of the operating system of the computer are not covered. Therefore, the expressiveness of periodic TM packets and their time stamp evaluation is limited. The test-framework needs to run on a real-time operating system to allow a realistic investigation of the temporal behaviour of the SUT.

Another limit is the implication that takes palace when executing a test case. If a specific Oracle state is reached, verified by the test-framework, it is expected (implied) that all the prior addressed TTC states are correct. The assumption is that all transmitted TTC packets are without any failure. For the programmer it is hard to realize that the expected TTC states was not reached, when the Oracle state is verified. What actually happens in such a case is that a wrong trace in the Kripke structure was tested, which has the expected Oracle state as final state. Additionally to that, it is not clear if the test case itself is correct. This is assumed within the test-framework and a general problem in testing.

A similar limitation resides in the system model. The structural tests reveal only that the structure of the system model fits the OBSW structure. But the model of the structure itself can be wrong, as it is not tested.

Also problematic is the potential loss of precision when the OBSW or the TEEP tries to store a floating point number in multiple $uint8_t$ data types when serializing the packet for the transmission. The workaround for this problem is a programmer-defined delta value that enables the comparison of floating point values. The comparison takes palace while the evaluation of the TM packet against the Oracle packet and is stated as successful if the difference is less than the defined delta value.

Another potential source for failures is the transformation of the system and the test model in compilable C++ code. There is always the threat of a flawed implementation of the code generator. Software has bugs and the

test-framework is no exception. Therefore, the programmer has to be aware that the translation from the model to the C++ code might be corrupted.

Chapter 8

Conclusion

8.1 Summary

The test-framework bridges the knowledge gap between the domain expert and the programmer when developing OBSW for satellites. The domain expert is able to define a system model of the OBSW, illustrating the software interface that is controlling his or her satellite component. This system model serves as an input for the test model, in which test cases can be defined. The domain expert is able to write multiple test cases to simulate a realistic operation of his or her component. The test case input and expected output data are instantiated TC & TM packets of the system model in the test model. These instantiated packets are Test Telecommand and Oracle packets. The test cases and the data packets, are translated via a code generator in compilable C++ code. Additionally, the basic structure of the OBSW and the on-board routing mechanism (CDH subsystem) of the packets is converted into C++ classes. The entire code is integrated automatically in the Test Execution & Evaluation Platform. To determine the coverage of the test suite, the domain expert is supported by a Coverage Report that investigates the structural coverage of his or her system model in the test model. When the coverage is stated as sufficient by the domain expert, his or her work is done and handed over to the programmer.

The programmer begins to implement the OBSW of the satellite. The OBSW is the SUT and is connected, via a ringbuffer as communication interface for the packet transmission, to the Test Execution & Evaluation Platform. During the implementation of the OBSW, the programmer receives immediate feedback on the correctness of the OBSW under development by the execution of the test cases. He or she is able to determine the accurateness of the OBSW's behaviour by an investigation of various logs and test case evaluation reports. In addition

to that, the coloured evaluation result is forwarded to standard out. If the programmer detects an error in the system or test model, he or she is able to discuss this with the domain expert on the basis of the models, the logs and the reports.

8.2 Outlook

There are a plethora of future applications and additional features for the test-framework. A potential future scenario is the enrichment of the test-framework by a feature, which is calculating the communication traffic between the base station and the satellite over time. Heavily communicating systems could be revealed at an early stage of the development, facilitating the comparison against the planned communication budget of each system of the satellite.

Additionally, the usage of the test-framework for PiL and HiL tests is also possible. The hardware connection to the on-board computer has to be established by actual hardware and correlating hardware drivers. Exceptional situations which might occur could be simulated and functioning workarounds already implemented in the OBSW before an event actually triggers the same when the satellite is operating in space.

It is also possible to enable early simulation of the entire system. Therefore, an environment model is needed in addition to the test and system model. The environment model simulates the space environment as well as the technical behaviour of the satellite. This model could be used to generate behavioural C++ code, which can be attached to the SUT to simulate the entire system during the development of the OBSW, respectively the execution of the test cases.

The test-framework uses the same interface as the ground station to the satellite. Therefore, it could be used to write TC procedures which can be transmitted to the operating satellite via the base station. These procedures can be generated before the mission starts and carried out when the satellite mission begins.

To go one step further, the test-framework itself could be integrated in the ground station, to enable a dynamic reaction on specific TM packets that arrive from the satellite. Especially in early mission phases, this would shift control over the satellite from the on-board computer to the ground station. Early mission phases are all phases before the satellite reaches its final working environment. Nowadays, the satellite mostly manages itself during those phases,

leaving mission control out of the decision process, because the evaluation of the TM packets is primarily conducted in a manual fashion.

Finally, the test-framework can be used to perform mission online tests. The actual TM packets of the OBSW have to be forwarded to the test-framework, in which they are compared to specific Oracle packets. This would facilitate the detection of errors in the satellite's hardware components, when expected TM packets fail the evaluation. In case an error is detected, the test-framework can be used to build TC packet procedures to find a workaround for the error. These procedures could then be forwarded directly to the satellite and the result mission-online evaluated.

Bibliography

[1] Space Foundation. *Space Foundation - Research and Analysis - Space Foundation Indexes*. URL: http://www.spacefoundation.org/programs/research-and-analysis/space-foundation-indexes, accessed: [25/May/2015].

[2] CubeSat. *Homepage - CubeSat*. URL: http://www.cubesat.org/, accessed: [15/Apr/2015].

[3] SpaceX. *Homepage - SpaceX*. URL: http://www.spacex.com/, accessed: [15/Apr/2015].

[4] Space Foundation. *The Space Report 2014*. Space Foundation. USA, Colorado Springs. 2014. ISBN-13: 978-0-9789993-7-7

[5] OECD. *The Space Economy at a Glance 2014*. OECD Publishing. URL: http://dx.doi.org/10.1787/9789264217294-en. November 2014.

[6] ESA Board for Software Standardisation and Control. *Guide to software quality assurance*. ESA PSS-05-11 Issue 1 Revision 1. PARIS CEDEX, France. March 1995.

[7] European Cooperation for Space Standardization. *Homepage-ECSS*. URL: http://www.ecss.nl/, accessed: [19/April/2015].

[8] *Space project management - Project planning and implementation*. European Cooperation for Space Standardization, Requirements & Standards Division. Noordwijk, Netherlands. 2009.

[9] Wilfried Ley, Klaus Wittmann, Willi Hallmann. *Handbuch der Raumfahrttechnik*. Carl Hanser Verlag. 2008.

[10] Lui Sha, Rajkumar, R.; Lehoczky, J.P., *Priority inheritance protocols: an approach to real-time synchronization*, Computers, IEEE Transactions on , vol.39, no.9, pp.1175,1185, Sep 1990.

[11] Mike Jones. *What really happened on Mars?*. Record. URL: http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html. December, 1997.

[12] S. McConnell. SM 2004. *Code Complete: A practical Handbook of Software Construction*. 2nd. Microsoft Press. Redmond, Washington, USA.

[13] Paul K. Martin. *Review of NASA'S Lessons Learned Information System*.Report NO. IG-12-012 (Assignment NO. A-11-010-00) .National Aeronautics and Space Administration.Washington, DC, USA. March, 2012.

[14] Deshmukh, Weps, Isidro, Gerndt. *Model Driven Language Framework to Automate Command and Data Handling Code Generation*.Brunswick, Germany. January 2015.

[15] A. Håkansson, *Portal of Research Methods and Methodologies for Research Projects and Degree Projects*. WORLDCOMP'13 - The 2013 World Congress in Computer Science, Computer Engineering, and Applied Computing, 22-25 July, 2013 Las Vegas, Nevada; USA.

[16] S. Keshav. 2007. *How to read a paper*. SIGCOMM Comput. Commun. Rev. 37, 3, 83-84, July 2007.

[17] European Cooperation for Space Standardization. *Space engineering Ground systems and operations — Telemetry and telecommand packet utilization*. ESA Publications Division. Noordwijk, Netherlands. January 2003.

[18] Mark Utting, Bruno Legeard. *Practical Model-Based Testing: A Tools Approach* Morgan Kaufmann Publishers Inc. San Francisco, CA, USA. 2007.

[19] Lee Copeland. *A Practitioner's Guide to Software Test Design* Artech Hous Publishers. Norwood, USA. January 2004.

[20] Seon-Jae Jang; Hae-Geun Kim; Youn-Ky Chung, *Manual Specific Testing and Quality Evaluation for Embedded Software*. Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on , vol., no., pp.502,507, 14-16 May 2008.

[21] Thopate, H.; Kachewar, R.R.. *Chameleon model based automation framework design for testing in agile environments*. Software Engineering (CONSEG), 2012 CSI Sixth International Conference on , vol., no., pp.1,4, 5-7 Sept. 2012.

[22] M. Fewster, D. Graham. *Software Test Automation - Effective use of test execution tools*. Addison-Wesley. 1999.

[23] Xinbian Wang; Guangjun He. *The research of data-driven testing based on QTP*. Computer Science & Education (ICCSE). 2014 9th International Conference on . vol., no., pp.1063,1066. 22-24 August 2014.

[24] M. Voelter. *DSL Engineering Designing, Implementing and Using Domain-Specific Languages*. self-published (dslbook.org). 2013.

[25] W. Beaton. *Eclipse Platform Technical Overview (v3.1), April 2006*. URL: https://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html, accessed: [03/May/2015].

[26] Maximilian Koegel, Jonas Helming. *What every Eclipse developer should know about EMF*. April 2015. URL: http://eclipsesource.com/blogs/tutorials/emf-tutorial/, accessed: [03/May/2015].

[27] Prof. Gerti Kappel. *Eclipse Modeling Framework* Technical Univaersity Dresden. April 2012. Lecture-notes: Real-Time Systems Lab . URL: http://www.es.tu-darmstadt.de/fileadmin/download/lehre/mbse/LectureSlides/02-EMF.pdf, accessed: [03/May/2015].

[28] *Documentation: Package org.eclipse.emf.ecore*. URL: http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html,accessed: [03/May/2015].

[29] S. Efftinge, M. Voelter. *oAW xText: A framework for textual DSLs*. Version 1.1. September 2006.

[30] *Xtend User Guide*. URL: https://www.eclipse.org/xtend/documentation/2.4.0/Documentation.pdf . April 2013.

[31] Leslie Lambort. *What good is temporal logic*. Elsevier Scinece Publishers B. V. (North-Holland). Computer Science Laboratory, Manlo Park, USA. 1983

[32] Klaus Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. Springer Science & Business Media. 2004.

[33] Pierre Wolper. *Temporal Logic Can Be More Expressive* Academic Press, Inc. Computer Science Department, Stanford University, Stanford, California, USA. 1983.

[34] Cesar Sanchez, Martin Leucker. *Linear temporal logic with until and next, logical consecutions* Elsevier. Department of Computing and Mathematics, Manchester Metropolitan University, Manchester, UK, Siberian Federal University, Krasnoyarsk, Russia. May 2008.

[35] V. Rybakov. *Regular Linear Temporal Logic with Past* Springer-Verlag Berlin Heidelberg. Madrid Institute for Advanced Studies (IMDEA Software), Spain. Spanish Council for Scientific Research (CSIC), Spain. Technische Universitaet Muenchen, Germany. 2010.

[36] Christel Baier, Joost-Pieter Katoen. *Principles of Model Checking* MIT Press, Cambridge, England. URL: http://is.ifmo.ru/books/_principles_of_model_checking.pdf . 2008

[37] Virtual Satellite Project. DLR - Institute for Software. URL: https://software.dlr.de/p/virsat/home/ accessed: [01/06/2015]

[38] Autonomous Terrain based Optical Navigation. DLR - Institute of Space Systems. URL: http://www.dlr.de/irs/en/desktopdefault.aspx/tabid-6657/10924_read-24821/, accessed: [01/06/2015]

[39] Nan Li, Jeff Offutt. *A Test Automation Language Framework for Behavioural Models*. IEEE Eight International Conference on Software Testing. Verification and Validation Workshop (ICSTW). 12th Workshop on Advances in Model Based Tsting (A-MOST 2015) New York, USA. 2015

[40] Iyenghar P., Pulvermueller, E., Westerkamp, C., Wuebbelmann, J.. *Integrated model-based approach and test framework for embedded systems*. Specification and Design Languages (FDL). 2011 Forum on , vol., no., pp.1,8, 13-15 Sept. 2011

[41] Shaoying Liu. *Automatic Specification-Based Testing: Challenges and Possibilities*. Theoretical Aspects of Software Engineering (TASE). 2011 Fifth International Symposium on vol., no., pp.5,8, 29-31 Aug. 2011

[42] Andrew J. Dunar, Stephen P. Waring. *Power to Explore: A History of Marshall Space flight Center 1960-1990*. CreateSpace Independent Publishing Platform. URL: http://history.msfc.nasa.gov/book/. July 2012.

[43] Committee on Science and Technology House of Representatives. *Investigation of the Challenger accident*.U.S. Government Printing Office .Washington, DC, USA. October, 1986.

[44] *Code of Ethics for Engineers*. National Society of Professional Engineers. Alexandria, Virginia, USA. January 2003.

[45] IEEE Board of Directors. *IEEE Code of Ethics*. Institute of Electrical and Electronics Engineers (IEEE). URL: http://www.ieee.org/about/ieee_code_of_conduct.pdf. June 2014.

[46] Justyna Zander, Ina Schieferdecker, Pieter J. Mosterman *Model-Based Testing for Embedded Systems*. CRC Press. September 2015.

[47] Mark Utting,Alexander Pretschner, Bruno Legeard. *A TAXONOMY OF MODEL-BASED TESTING*. The University of Waikato. Hamilton, New Zealand. April 2006.

[48] Justyna Zander-Nowicka. *Model-based Testing of Real-Time Embedded Systems in the Automotive Domain*. Doctoral Dissertation. Technical University Berlin. 2009.

Appendix A

Introduction

Mission phases milestones

Table A.1 displays the milestones of the space mission life cycle form the European Cooperation of Space Standardization.

Abbreviation	Milestone
MDR	Mission Definition Review
PRR	Preliminary Requirements Review
SRR	System Requirements Review
PDR	Preliminary Design Review
CDR	Critical Design Review
QR	Qualification Review
AR	Acceptance Review
ORR	Operational Readiness Review
FRR	Flight Readiness Review
CRR	Commissioning Result Review
LRR	Launch Readiness Review
ELR	End-Of-Life Review
MCR	Mission Close-Out Review

Table A.1 Mission phases milestones. Source: [8]

Satellite sizes

Table A.2 displays the different satellite classes, categorized regarding their mass.

Satellite class	Mass
Conventional satellite	> 500 kg
Mini-satellite	100 ... 500 kg
Micro-satellite	10 ... 100 kg
Nano-satellite	1 ... 10 kg
Pico-satellite	0.1 ... 1 kg

Table A.2 Satellite classification according to the mass. Source: [9, p: 666]