Model based requirements management for the space mission design framework

Master's Thesis

Eren Bellisoy

January 22, 2019

Institute of Software Engineering and Automotive Informatics
at
Technische Universität Carolo-Wilhelmina in Braunschweig (Germany)

Dr.-Ing. Christoph Seidl

Abstract

Requirements engineering plays an important role both in software and systems engineering. It is the process of defining, documenting, and maintaining requirements. Requirements traceability is a branch of requirements engineering, which establishes relationships between requirements and design artifacts, implementation artifacts, and test cases. Traceability provides several benefits both in software and systems engineering, one of them is to provide change impact analysis. When a particular requirement changes, it is implied that the artifacts related with the requirement should also change. In this thesis, we provide a new methodology for requirements management and traceability. Although the methodology is applicable for different systems engineering domains, space mission requirements and spacecraft models are the main focus of this thesis. The methodology consists of three parts. In the first part, we cover traceability between requirements and model artifacts. Unlike existing traceability approaches, our methodology provides automatic validation. In the second part of the methodology, we introduce modeling on the basis of requirements. The last part of the methodology covers requirements-based artifact reuse. Besides our theoretical contribution, we provide a prototype tool implementation which includes the features of the methodology. In our evaluation, we demonstrate our contribution by integrating our tool with Virtual Satellite, a spacecraft modeling software developed by German Aerospace Center (DLR).

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

A requirement is a condition or capability needed by a user to solve a problem or achieve an objective [1]. A set of requirements is used to define the capability and functionality of a product. This set of requirements, which is called requirements specification document, acts as a medium between stakeholders and developers. Moreover, it is also responsible for defining the purpose and functionality of the product. The success of a product is determined regarding to which degree it meets the purpose for which it was intended [2]. In other words, success of a product is determined by, what percentage of the requirements are fulfilled. A complete product should satisfy all of the requirements. The completenes of a product is measured by verification and validation [3].

A survey done by Standish Group states that 39% of software projects conducted in 2012 were successful, 18% of them have failed and 43% of them are challenged [4]. Mandal et al. state that 60% – 80% of those project failures can be attributed directly to poor requirements gathering, analysis, and management [5]. Boehm et al. state that requirements elicitation and requirements volatility are the two main factors that affects the cost and therefore the success of the project [6]."Requirements Engineering" addresses this problem. Méndez et al. define requirements engineering as "Requirements engineering (RE) aims at the discovery and specification of requirements that unambiguously reflect the purpose of a software system" [7]. Badly written requirements eventually lead to bad implementation and, therefore, the failure of the project. In order to address this issue, several standards are developed for writing good requirements. Moreover, limiting the usage of the natural language in the requirements and forcing a pattern to write the requirements is another approach to increase requirement quality. However, only well and unambiguously written requirements are not enough for the success of the project. Typically, the requirements specification document is a living document, where it is constantly subject to changes. Requirements volatility refers to additions, deletions and modifications of requirements during the system development life cycle [8]. In their survey, Zowghi and Nurmuliani [9] found out that the completion of a project on time and on budget strictly depends on the stability of the requirements. Similarly, from the survey conducted by Curtis et al. [10] fluctuating requirements came out as one of the most important challenges in developing large systems. Efficiently handling those changes is required since it may lead to redundant implementation or inadequate implementation which increases the cost and the delay of the project. Handling those changes is called "requirements volatility management" [8].

Requirements traceability is a common method to handle the modifications on the requirements [11]. It is the activity of creating and recording links between individual requirements and related artifacts such as use cases, design documents, code chunks or test cases. With the proper tracing, it is possible to analyze the impact of a change, and detect the artifacts which are possibly affected by the change on the requirement.

In the recent years, model driven engineering[12] became popular. With model driven engineering, the complexity of a system can be captured in an abstract model, where internal checks can be applied to detect and prevent many errors early in the life cycle of the system. Moreover, de-

sign information can be shared easily through the model. For the projects which use model driven engineering, traceability links can be used in order to establish the relation between the model elements and requirements. Although there are different approaches to trace the requirements to model artifacts [13], the existing approaches do not consider the state of the model artifact, whether the artifact satisfies the requirement or not. This lack of consideration creates two problems. First one is, in a scenario where a requirement is traced to multiple model artifacts, if the requirement changes, regardless of the change, all of the traced artifacts should be reviewed by the engineer, which is time consuming. Secondly, if some violation is missed, the only way to detect it is in the test phase, where design and development are already complete. Failure of tests forces engineers to revisit their work, which increases the project's cost. Detecting violations in the early phases of the project would save time for the developers and would reduce the cost of the project. In order to determine if the model artifact satisfies the requirement or not, human observation is necessary for many requirements. However, in a requirements specification document, some requirements contain quantifiable information. For example, in space mission requirements, this information can be the upper limit for the volume of some component of the spacecraft or the lower limit for the data transfer rate of some component. These requirements have the potential to be automatically validated during the modeling phase, when proper traceability links between the model artifact and the requirement exists. For this thesis the term *validation* is used to address that if a requirement is satisfied by the model artifact. The ambiguity with the terms validation and verification is discussed in Subsection 2.1.1. Validation defitinition by IEEE is "Confirmation by examination and provisions of objective evidence that the particular requirements for a specific intended use are fulfilled" [14]. With this automatic validation, any violation can be detected during the design phase, before implementing the actual product. Moreover, this automatic violation would also reduce the time needed for change impact analysis. As an example, when a change occurs on a requirement which is traced to multiple artifacts, it would be enough to review the artifacts which are violating the requirement, instead of reviewing all the artifacts which are traced to the requirement.

Thesis Goals

The goal of this thesis is to introduce a new methodology for tracing and validating requirements. As opposed to existing approaches, we will make validation a part of development phase. Moreover, with this thesis we aim to improve the state of the art in requirements volatility management. Although our methodology and solution should be applicable to a wide range of domains, we focus on space mission requirements and spacecraft design, and demonstrate the capabilities and improvements of our approach. Besides the theoretical contribution, we implement a prototype with the following capabilities;

- Ability to create traceability links between requirements and model artifacts.
- Automatic notifications on model artifacts when a requirement change occurs.
- Automatic validation mechanism applicable for requirements which contain quantifiable information.
- Ability to generate model artifacts on the basis of requirements.
- Ability to generate structured requirements.

- Ability to determine the re-usable model artifacts in different projects.

Organization

In Chapter 2, we give relevant information on the topics covered in this thesis, such as fundamentals of requirement, space mission requirements, and existing requirement management tools. In Chapter 3, we explain our scientific contribution for requirements management and automatic validation.Chapter 4 covers the design and architecture of our prototype implementation. In Chapter 5, we evaluate our contribution and discuss our results. In Chapter 6, we analyze and give an overview of related work in the field of requirements management and traceability to highlight our contribution. In Chapter 7, we provide a brief conclusion our work and summarize this thesis. We elaborate on our ideas about future work in Chapter 8.

2 Background

In this chapter, we cover the necessary background information for this thesis. Relevant aspects of requirements are explained in Section 2.1. In Section 2.2, we give details on the concept of traceability, its usage and limitations. In Section 2.3, we give an overview about space missions, space mission requirements, and spacecraft modeling. In Section 2.4, we analyze the existing commercial tools for requirements traceability and their limitations.

2.1 Requirements Fundamentals

The quality of the requirements is one of the leading factors for the success of a project [5] [6]. The importance of requirement qualitly led many research to be conducted in order to define the standarts for writing good requirements[16][17][18][15]. The example structure of the requirements can be seen in Table 2.1. Good requirements should be written according to following criteria;

- **Avoiding ambiguities:** The ambiguous terms used in the requirements makes it hard to understand.
- **Avoiding negative statements:** If it is possible to give the positive statement, the negative statements should not be used.
- **Avoiding multiple information:** Requirements should be atomic. One requirement should avoid providing multiple information.
- **Consistency in requirements:** The requirements should be consistent with each other. Any pair of the requirements should not be conflicting.
- **Avoiding speculation:** Requirements should not use speculative wording. The following words should be avoided; usually, generally, often, normally, typically.

[Condition] [Subject] [Action] [Object] [Constraint]
EXAMPLE: When signal x is received **[Condition]**, the system **[Subject]** shall set **[Action]** the signal x received bit **[Object]** within 2 seconds **[Constraint]**.
Or
[Condition] [Action or Constraint] [Value]
EXAMPLE: At sea state 1 **[Condition]**, the Radar System shall detect targets at ranges out to **[Action or Constraint]** 100 nautical miles **[Value]**.
Or
[Subject] [Action] [Value]
EXAMPLE: The Invoice System **[Subject]**, shall display pending customer invoices **[Action]** in ascending order **[Value]** in which invoices are to be paid.

Table 2.1: Syntax of the requirements [15]

- **Avoiding indefinable terms:** Requirements should not use indefinable terms such as the following ones; user-friendly, versatile, flexible, approximately, as possible, efficient, improved, high performance.

Requirement Attributes

Requirements should have descriptive attributes in order to ease the understanding as well as management. The examples of those attributes are;

- **Identification** Each requirement should have a unique identifier. Identifier of the requirement should stay the same even if the requirement changes.
- **Stakeholder Priority** Each requirement should have a priority such as high, low or medium. This attribute is important to make decisions regarding design decisions and project planing, such as what to develop first and how much resource should be allocated for implementation of a requirement.
- **Source** Requirements may be written by different people. In order to change a requirement, its originator should be consulted. Finding the originator of the requirements is easier if the requirements are specifying their originator.

Requirement Boilerplates

Requirements can be written in natural language. However, such requirements can be ambiguous and can be understood differently by different people. Furhthermore, extracting information automatically from the natural language requires advanced techniques such as *Natural Language Processing* [19][20]. However these techniques are not 100% reliable as we discuss in Chapter 6. In order to address this issue, requirement boilerplates [21][22] are introduced. Requirement boilerplates limit the usage of the natural language, and provide a syntactical template to write the requirements. Requirements written according to the template are called *Boilerplate Requirements*. These requirements can be parsed with computers more reliably than the requirements given in natural language. We will provide the boilerplates for space mission requirements in Section 3.6, and explain their strengths and weaknesses.

2.1.1 Verification vs. Validation

The terms verification and validation are hard to distinguish since they are similar with each other but not exactly the same. Boehm states that validation answers the question "Are you building the right product?" whereas verification answers the question "Are you building the product right?" [23]. The problem comes from the fact that both validation and verification processes are conducted by tests. In our methodology, guaranteeing that a component is built accordingly to the requirements can mean both that the product is being built right and the right product is being built. Another definition from Sharma is verification : "Did I build what I need?" and validation : "Did I build what I said I would?" [24].

In our methodology we can only guarantee that we build the product according to the requirements. However, we cannot guarantee that what we are building is useful for the stakeholders or it is needed in general. With those considerations in mind, we have named our process "automatic validation" instead of "automatic verification".

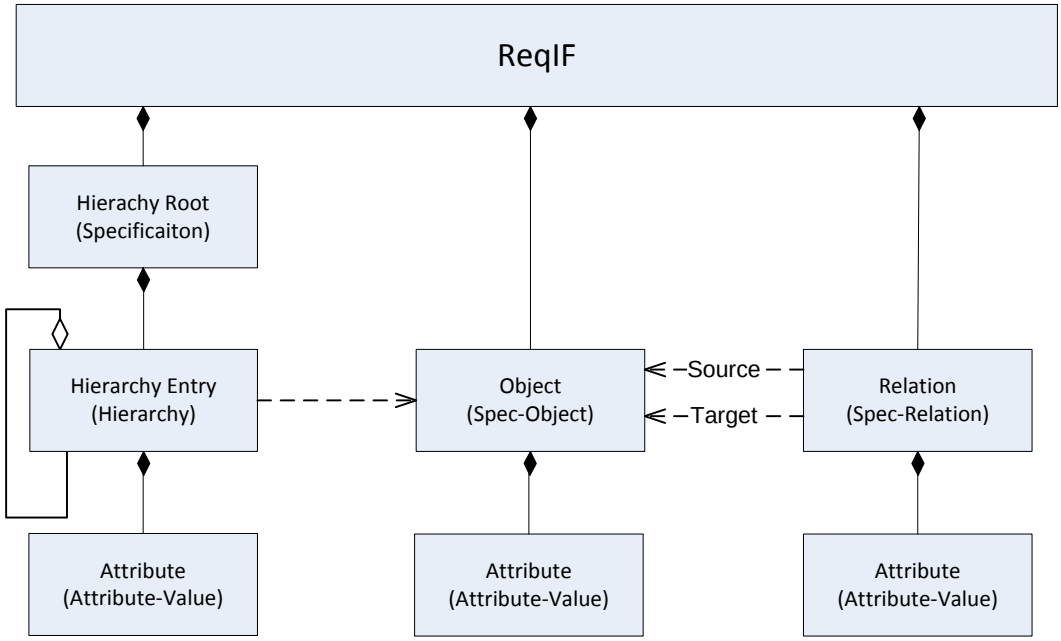


Figure 2.1: ReqIF format [27]

2.1.2 Requirements Interchange Format

The Requirements Interchange Format(ReqIF) was created for the German automobile industry in 2003 [25]. German automotive companies typically work with hundreds of suppliers. This situation requires requirements to be exchanged between parties. The problem was the loss of data when exporting a subset of requirements to send them to a subcontractor. DOORS allow requirements exchange between different parties without data loss, however, both parties need to have DOORS. Since ReqIF is an international Object Management Group (OMG) standard, ReqIF allows exchanging product requirements among companies without information loss. Moreover a particular software is not needed, parties can use different software or even develop their own software. ReqIF data contains three types of data; Objects, Relations and Hierarchy-Entries as depicted Figure 2.1. Those can have custom attributes as well, such as id, date, description and owner. The attributes are not static, they can be configured by the user. Objects are the actual requirements. Hierarchy-Entries are used to group similar requirements. A set of requirements related to a particular component of the system or requirements which have same attributes or requirements with the same owner can be grouped under a spec hierarchy. Relations are used to link requirements with each other. Different relation types with different attributes can be configured by the user. More information regarding the standard can be found at [26].

The situation in the aerospace industry is similar with automobile industry. The producers of satellites and planes also work with many suppliers. Since a dependence on a commercial third party software is not desired, the ReqIF standard will be used as a basis for this thesis.

2.2 Traceability in Practice

Traceability has different definitions in the literature. Ramamoorthy et al. describe traceability as "Traceability refers to the ability of tracing from one entity to another based on given semantic relations" [28]. Gruia-Catalin Roman describes it as "Traceability refers to the ability to cross-reference items in the requirements specification with items in the design specification" [29]. Lago et al. defines traceability as "Traceability is the ability to describe and follow the life of a software artifact and a means for modeling the relations between software artifacts in an explicit way" [30]. In general, two types of traceability exist: vertical and horizontal traceability. Vertical traceability is the relation between hierarchical entities such as requirements to design to code. Horizontal traceability is relation between non-hierarchical entities such as relations among requirements. However these terms are used interchangeably in different papers. Anquetil et al. [31] instead uses the terms, inter and intra traceability to clear the ambiguity. Inter traceability: A relationship between two artifacts with different levels of abstraction such as requirements and software artifacts. Intra traceability: the relationship between two artifacts that are on the same level of abstraction such as the relationship between related requirements, between models, between software artfacts. In model driven engineering, traceability is used in the context of model transformations, for linking elements in different levels of granularity. Regarding the requirements traceability, Godel and Finkelstein define two types [32]. Pre-Requirement Specification traceability: indicates the origins of the requirements. Requirements may come from different stakeholders such as customers, project managers or developers. Pre-Requirements Specification traceability ensures that a requirement can be traced back to a person or a group. Post-Requirement Specification traceability: The traceability of the requirement to the software artifacts.

With this thesis, our main concern is to trace the requirements to the model artifacts. Pre-Requirement Specification traceability and traceability between different software components and model transformations are out of scope of this thesis. Although there are many tools dedicated for requirements management and traceability, they fail to address the problem we have mentioned in Chapter 1. Capabilities and limitations of the existing software are analyzed in Section 2.4.

2.3 Space Missions

Space missions are conducted in seven phases [33, 34]. The objectives of those phases are explained in Table 2.2 which is taken from NASA's system engineering handbook. However the phases and the purposes are not specific to NASA, ESA also uses the same phases for space missions.

Phase A is the phase where system engineers elicit the requirements. The success of the mission critically depends on this phase, since badly elicited requirements will lead to eventual failure. Our methodology starts from this point by providing guidelines for writing requirements which are possible to validate automatically.

Phase B is where the preliminary system design is conducted. In that phase, engineers from multiple different fields create the design of the each system component. The traceability methodology presented in this thesis aims to aid the engineers in this phase by providing ability to create traceability links between requirements and design artifacts, automatic validation, artifact generation on the basis of requirements, and support for re-usability of the previous design.

Phase	Purpose	Typical Outcomes
Pre-Phase A Concept Studies	To produce a broad spectrum of ideas and alternatives for missions from which new programs/projects can be selected. Determine feasibility of desired system, develop mission concepts, draft system-level requirements, assess performance, cost, and schedule feasibility; identify potential technology needs, and scope	Feasible system concepts in the form of simulations, analysis, study reports, models, and mock-ups
Phase A Concept and Technology Development	To determine the feasibility and desirability of a suggested new system and establish an initial baseline compatibility with NASA's strategic plans. Develop fnal mission concept, system-level requirements, needed system technology developments, and program/project technical management plans.	System concept defnition in the form of simulations, analysis, engineering models and mock-ups, and trade study defnition
Phase B Preliminary Design and Technology Completion	To determine the feasibility and desirability of a suggested new system and establish an initial baseline compatibility with NASA's strategic plans. Develop fnal mission concept, system-level requirements, needed system technology developments, and program/project technical management plans.	End products in the form of mock-ups, trade study results, specifcation and interface documents, and prototypes
Phase C Final Design and Fabrication	To complete the detailed design of the system (and its associated subsystems, including its operations systems), fabricate hardware, and code software. Generate fnal designs for each system structure end product.	End product detailed designs, end product component fabrication, and software development
Phase D System Assembly, Integration and Test, Launch	To assemble and integrate the system (hardware, software, and humans), meanwhile developing confdence that it is able to meet the system requirements. Launch and prepare for operations. Perform system end product implementation, assembly, integration and test, and transition to use.	Operations-ready system end product with supporting related enabling products
Phase E Operations and Sustainment	To conduct the mission and meet the initially identifed need and maintain support for that need. Implement the mission operations plan.	Desired system
Phase F Closeout	To implement the systems decommissioning/disposal plan developed in Phase E and perform analyses of the returned data and any returned samples.	Product closeout

Table 2.2: Space Mission Phases [33]

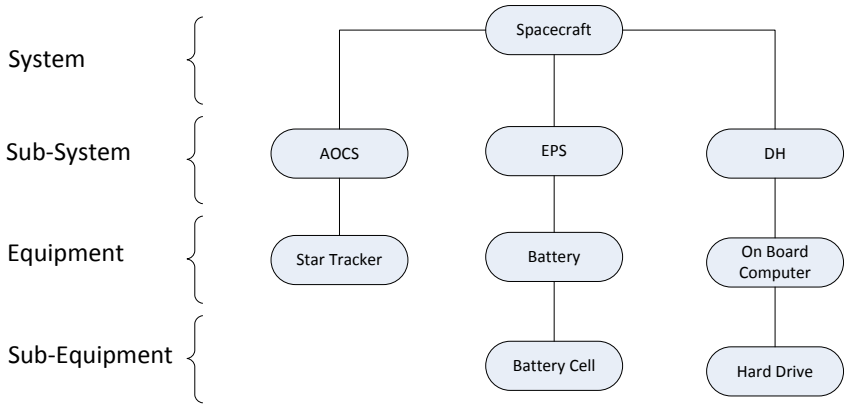


Figure 2.2: The structure of a spacecraft

Terminology for the Spacecraft Design

A space mission has three segments namely; space segment, launch segment and ground segment[35]. Space segment is the actual spacecraft. The spacecraft is considered as a *System*. The spacecraft contains a payload, which is the cargo to be sent to the space. As an example, a telescope can be carried in the payload. For the manned missions, payload carries the astronauts. Besides payload, spacecraft has different sub-systems. Different sub-systems provides different functionalities such as; providing orbit control, providing data storage, providing power. Each sub-system is composed of different equipment. Furthermore, some equipment are also composed of different sub-equipment. Figure 2.2 illustrates the general structure. During the design of a spacecraft, first system requirements are derived from mission requirements, then subsystem requirements are derived from system requirements, and finally equipment requirements are derived from sub-system requirements.

Space Mission Requirements

Software requirements, in general, do not contain much quantifiable information. Many implementation artifacts are responsible for satisfying a single requirement [36].Table 2.3 explains the general structure of mission requirements. As it can be seen in Table 2.3, in mission requirements most of the requirements are quantifiable and, in general, a particular mission requirement can be traced to a few components of the spacecraft. As an example, resolution requirements from the project FireSat II can be traced directly to the camera component of the satellite. This creates an opportunity to trace quantifiable values in the requirements to attributes of system components. With the help of traceability links, it is possible to validate the quantifiable information automatically.

Reusability in Space Missions

Space missions are not always one of a kind[37]. The existing hardware, design and the requirements can be reused in different missions. Writing the requirements for a mission is a manual work and it is also time consuming, therefore costly. Among different space missions, there is a huge probability that some of the existing requirements are similar. These requirements can be shared and reused. Similary, designing components for every mission is redundant, if one of the existing design can be

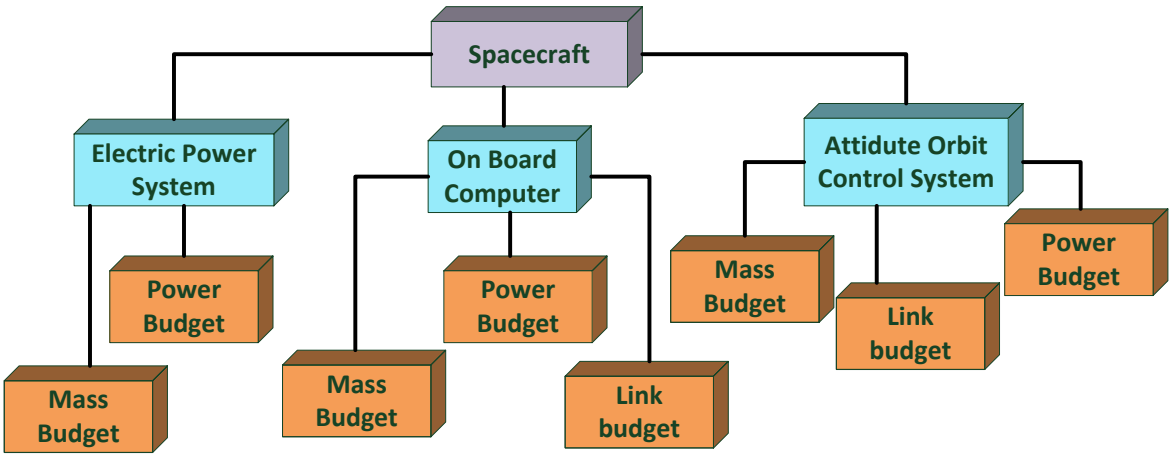


Figure 2.3: Budgets of the given components of a satellite

used. This reusability provides direct cost reduction for the mission. We provide our contribution on requirement reusability in Section 3.6. Moreover, our contribution for reusability of the existing desing is presented in Section 3.8.

2.3.1 Budgets of the Components

The components of a satellite receive budgets[38] from the mission requirements. System engineers define different budgets(mass budget, power budget and link budget) and distribute them to the sub-systems and equipment. Mass budget is calculated from the launching requirements, and then consumed by each component. E.g. if the carrier rocket can carry at most 20 kg, then the combined weight of the spacecraft should not exceed 20 kg. It is critical to no to exceed this budget, since the carrier rocket can fail during the launch otherwise. Power budget is calculated through the lifetime requirements of the mission. Link budget must be well distributed to the components to avoid network congestion within the spacecraft. Different systems and sub systems uses different budgets. An example can be seen in Figure 2.3. EPS does not have a link budget because it is not involved in data transaction. However, it consumes from the mass budget and power budget. These budgets put restrictions on the subsystems.

Small Satellite Technology Experiment Platform (S2TEP)

In order to increase the reusability in space missions, DLR is developing S2TEP [40] platform. The main idea behind S2TEP is to design spacecrafts as modular and scalable as well as cost effective. With this way, existing design components can be reused in the future missions. From the engineers in DLR Bremen, we were able to obtain real requirements for a S2TEP spacecraft. These requirements are used in this thesis to give examples and to ease the understanding of our methodology.

2.3.2 Model Driven Software Development

In software engineering UML (Unified Modeling Language) is a popular design approach to model classes, their attributes, the relations among classes, object interactions and workflow of the software. However, UML has the main purpose of design and documentation since the code and the

Requirement	Factors which Typically Impact the Requirement	FireSat II Example
	FUNCTIONAL	
Performance	Primary objective, payload size, orbit, pointing	0,12 sensivity at 300K 500 m resolution 500 m location accuracy
Coverage	Orbit, swath width, number of satellites, scheduling	Daily coverage of 750 million acres within continental US
Responsiveness	Communications architecture, processing delays, operations	Send Registered mission data within 30 min to up to 50 users
Secondary Mission	As Above	Land and sea surface temperature, high resolution water vapor imagery and crude winds over the continental US
	OPERATIONAL	
Duration	Experiment or operations, level of redundancy, altitute	Mission operational at least 10 yrs
Availability	Level of redundancy	98% excluding weather, 3-day maximum outage
Survivability	Orbit, hardening, electronics	Natural environment only
Data Distibution	Communications architecture	Up to 500 fire-monitoring offices +2,000 rangers worldwide (max. of 100 simultaneous users)
Data Content, Form, and Format	User needs, level and place of processing, payload	Location and extent of fire on any of 12 map bases, average temperature for each 30 m^2 grid

Table 2.3: Example of Top-Level Mission Requirements [39]

UML diagrams are not connected with each other. Model Driven Software Development(MDSD) addresses this issue by generating code directly from the model. This way any change on the model will be automatically reflected to the generated code [41].

Eclipse Modeling Framework (EMF)

EMF is a modeling framework based on Eclipse, which provides automatic code generation from a structured data model. The model is represented in XMI (XML Metadata Interchange). From the model, EMF generates Java classes for the model, editor classes for visualization, and adapter classes such as commands to make changes on the model [42]. We will use EMF to model our traceability links in our implementation.

2.3.3 Space System Engineering Software - Virtual Satellite

Virtual Satellite (VirSat) is an EMF based tool currently under development in DLR with the goal of supporting the whole lifecycle of spacecraft assembly and simulation. The software focuses on early stages of a space mission, mainly Phase A and B. The software is currenty used by engineers in Concurrent Engineering Facility (CEF) in Bremen. During those phases, engineers from different disciplines work together to create a preliminary satellite design. VirSat repsresents the spacecraft as a data model where it is composed of different subcomponents. With VirSat, instead of constantly exchanging design documents, all of the engineers are able to work on a model collaboratively. Although the software is used mainly for modeling a satellite, it also offers different features such as 3D visualization of the satellite, modeling the possible faults and fault trees for the satellite and eventually formal verification [43].

2.4 Capabilities of Requirement Management Tools

In this section, we analyze the available software support for requirements management. In Subsection 2.4.1, we analyze capabilities and limitations of commercial requirement management software. In Subsection 2.4.2, we analyze drawbacks of the Eclipse based tools since we implement our

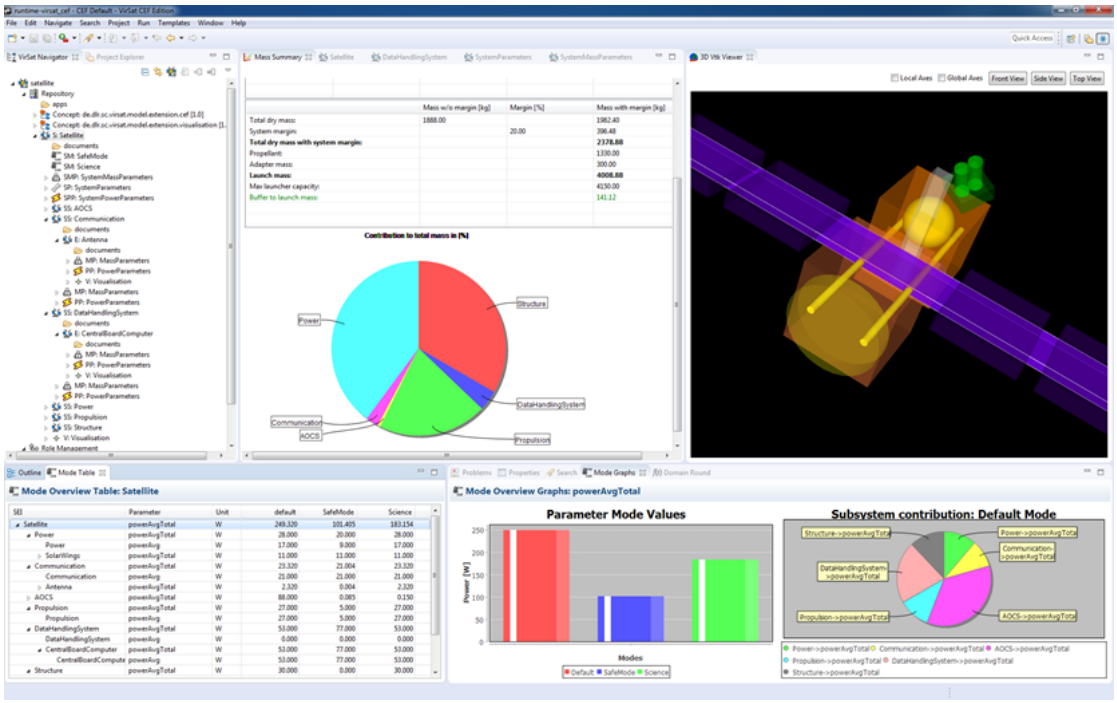


Figure 2.4: Space System Engineering Software - Virtual Satellite [44]

prototype with Eclipse.

2.4.1 Commercial Requirement Management Tools

It is possible to find different requirement management tools on the market. From a comparison made in August 2018 [45], we have selected twelve different tools and evaluated their capabilities and limitations. Detailed analysis can be seen in Table 2.4. We have analyzed five properties in parallel with our goals.

- **Traceability Links:** Is it possible to create traceability links between requirements and some other artifacts? Those artifacts can be use case diagrams, other requirements, classes, or other

	Traceability links	Customizable Traceability Link Types	Automatic Validation of Quantifiable Requirements	Change Impact Analysis	Artifact Generation on the Basis of Requirements
BluePrint	YES	YES	NO	YES	NO
Helix ALM	YES	YES	NO	YES	NO
InteGreat	YES	YES	NO	NO	NO
Visure	YES	YES	NO	YES	NO
Kovair	YES	YES	NO	NO	NO
Aha	YES	NO	NO	NO	NO
Avolution Abacus	YES	YES	NO	YES	NO
Aligned Elements	YES	YES	NO	NO	NO
Topteam Analyst	YES	YES	NO	NO	NO
Process Street	NO	NO	NO	NO	NO
Visual Trace Spec	YES	YES	NO	NO	NO
DOORs	YES	YES	NO	YES	NO

Table 2.4: Comparison of existing requirements management tools

files such as Word or Excel documents.

- **Customizable Trace Link Types:** Is it possible to define different types such as "related to" , "implements", "verifies".
- **Change Impact Analysis:** When a requirement change occurs, does the software provide notifications about which artifact may have been affected? This feature is only possible if the tool supports creation of traceability links.
- **Automatic Validation of Quantifiable Requirements:** Is it possible to trace quantifiable values in the requirements to the attributes of the artifacts?
- **Artifact Generation on the Basis of Requirements:** Is it possible to generate artifacts from the requirements with the traceability links? Instead of creating an artifact and then creating a trace link between the artifact and the requirement, with this feature the two steps would be merged into one.

Most of the requirements tools are also concerned with the project management. In general, they allow creation of use case diagrams, activity flows and their links to the requirements. However tools such as Process Street only focus on the management part and they do not provide traceability. The tool Aha! provides traceability links but without customization. The rest of the tools are capable of creating custom traceability links. However some of them do not provide change impact analysis.

DOORS is the leading requirements management tool in the market, used by famous companies such as BOSCH, Lockheed Martin and Philips [46]. However, even DOORS does not support automatic validation. Nevertheless, requirements can be traced to files, classes, diagrams. With the help of those traceability links DOORS is able to create a change impact report when a requirement change occurs which includes all the traced artifacts.

Tools such as Blue Print and Visure are similar with DOORS, they provide traceability between requirements and system artifacts, they offer change impact analysis and customizable traceability links. Automatic validation is not handled with any of the reviewed tools. Validation of the requirements is generally done by linking a requirement to a test case. These test cases can be run manually or via an automated script. As the problem we have addressed in Chapter 1, any requirement change requires re-run of the related test to validate the requirement again. We observe that none of the tools support artifact generation.

There are also specialized requirements management tools such as Aligned Elements. Aligned Elements focuses on medical device design. It has features especially designed to document the requirements, risks, reviews and tests.

2.4.2 Requirement Management Tools in Eclipse

Besides the commercial requirement management software, there has been some effort to develop requirements management tools in Eclipse. In this section, we explain them briefly. The eclipse based tools are;

RMF / ProR

Requirements Modelling Framework (RMF) [47] is a framework for working with requirements. It bases on ReqIF standard as explained in Subsection 2.1.2. The framework is an Eclipse-based open

source project. RMF provides the meta model for the requirements, whereas ProR provides the Graphical User Interface (GUI) to present and edit the requirements. By using the editors provided by ProR, users are able to create requirements, import from .reqIf files and can export them. It allows the creation of traceability links within requirements. However, it does not provide any feature for creating traceability links between requirements and EMF objects. In our implementation, we use RMF and ProR to model the requirements. Implementation details are discussed in Chapter 4.

Capra

Capra [48] is an Eclipse-based traceabillity management tool, which focuses on creating traceability links between different artifacts. Artifacts can be EMF elements or class files. It allows defining new traceabiliy link types and it also provides a visualization editor. Moreover, traceability links can be traced for tickets and bugs managed by Eclipse Mylyn. The downside of Capra is that, the tool does not provide any change impact analysis. This part is crucial to achieve our automatic validation goal. Since there is no change impact analysis, Capra also does not support automatic validation. Artifact generation on the basis of requirements is also not supported.

ReqCycle

ReqCycle [49] is an Eclipse-based tool developed for requirements management and traceability. Users are able to create requirements and import a set of requirements from different file formats such as .reqIf , .docx, .odt, .xslx. Requirements which are modeled with RMF are also supported. From those requirements, ReqCycle can create traceability links to model elements or class files. Creation of different types of traceability links are supported. Traceability matrices can be created to show the amount of requirements which are traced. As its drawbacks, ReqCycle does not provide change impact analysis. ReqCycle also does not provide any validation mechanism which is addressed by our initial problem statement. Artifact generation on the basis of requirements is also not possible.

3 A Methodology for Requirements Management

In this section, we present our methodology for requirements management. An overview of our methodology can be seen in Figure 3.1. In Section 3.1 we explain the structure of our traceability model and how we trace the requirements to model artifacts. In Section 3.2, we discuss the properties of requirements which can be automatically validated. Next, we introduce the validation engines, which are the core component of our automatic validation in Section 4.3. The methodology for tracing the requirements, which are not possible to validate automatically is explained in Section 3.4. The management of the requirements and respected traceability links are defined in Section 3.5. As explained in subsubsection 2.3, reusability of the requirements and design are two important factors for cost reduction . We provide our contribution on requirements sharing in Section 3.6, by analyzing the usage of structured requirements, their advantages and shortcomings and the methodology for creating such requirements. In Section 3.7 we describe the second step of our methodology, creating model artifacts on the basis of requirements. For the final step of our methodology, we present our contribution on the re-usability of the existing model artifacts in newer models by using our automatic validation technique, in Section 3.8.

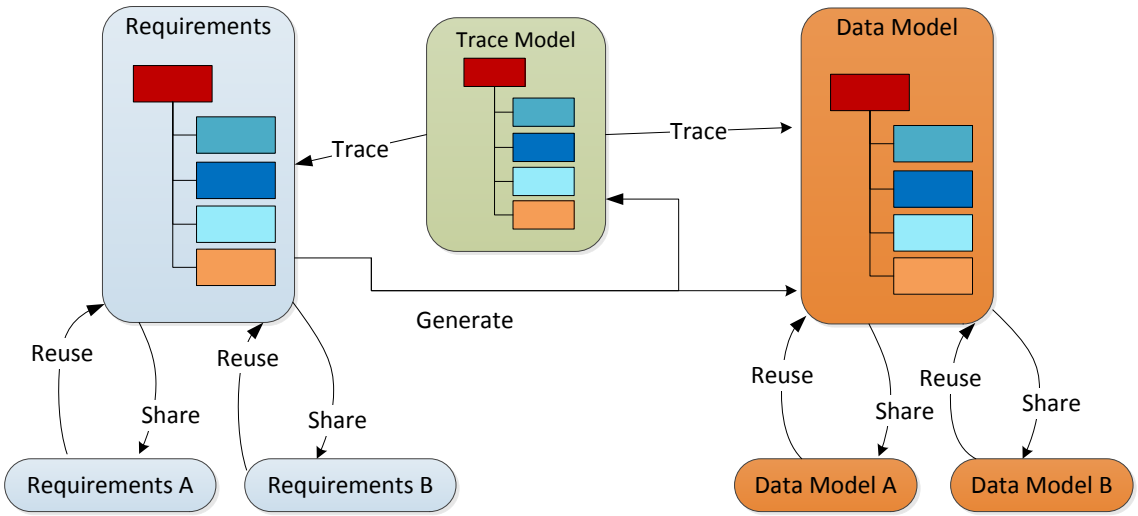
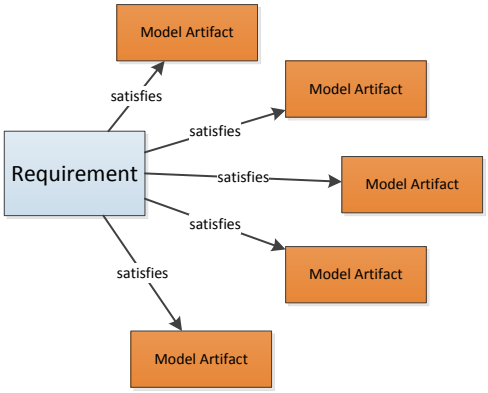
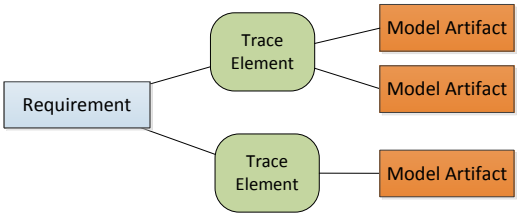


Figure 3.1: Overview of the Methodology



(a) The existing approach on traceability



(b) Our traceability model

Figure 3.2: Different traceability methodologies

3.1 A New Traceability Model

The existing requirements traceability approaches consider simple traceability links as it can be seen in Figure 3.2a. These links trace the requirements to model artifacts with little semantic information. With this lack of semantics, it is challenging to perform automatic validation. Instead of such simple traceability links, we define a new traceability model. In our model, the traceability links are not simple links, instead we introduce the *Trace Elements*. The trace elements are the middle nodes between the traced artifacts and the requirements. Each of them have a name and a description and a validation engine to perform automatic validation. The trace elements have a source, which is the requirement, and one or more targets, which are the model artifacts. The trace elements are responsible for defining which artifacts are related with which requirement. The structure of our trace elements can be seen in Figure 3.2b.

3.2 Automatic Validation

In order to automatically validate a requirement, the information and the semantics presented in the requirement should be converted into machine-understandable data. In order to extract the information, we require the requirements to be atomic, one information per requirement and unconditional. Challenges of the non-atomic requirements and conditional requirements will be explained in Subsection 3.4.1. However, not every requirement is atomic. For the non atomic requirements, the first step for the automatic validation is to convert the existing requirements to atomic requirements. As an example, we use a requirement from the S2TEP project listed in Table 3.1. In that example, the requirement contains three different pieces of information regarding the voltage usage. Moreover, the changes to be made in the future are also included in the requirement. The requirement can be rewritten without losing any information as in Table 3.2. In Table 3.2 each requirement provides one information. However, extracting the information from the requirement is still a challenging task since the requirement is given in natural language. As explained in subsubsection 2.1, extracting the information from the requirement, or directly converting the requirement to a machine-understandable format would require advanced techniques such as *Nat-*

CUS-EPS-16	assembly	platform	applicable
All units onboard the S2TEP satellite connected to the unregulated battery bus shall operate nominally with full performance over the following DC voltage interface (measured at the unit power interface): 8 V as minimum voltage 10.0 V as nominal voltage 12.8 V as maximum voltage The above values are the mean voltage values, and are excluding noise, ripple and voltage spikes).			
Derived from the S2TEP-EPS characteristics. Note: Be aware that this requirement will change with later S2TEP missions to following values: 11.0 V as minimum voltage 14.0 V as nominal voltage 17.0 V as maximum voltage			
Verified by test			
Payload-relevant (the same applies to payloads)			

Table 3.1: Example Requirement From S2TEP Project

ural Language Processing [19][20]. The problem regarding those advanced techniques is that, they are not mature enough to meet the high standards in spacecraft system design. Extracted information is not always 100% correct. Which means that, with a large set of requirements, it is highly likely that, some of the information presented in the requirements will be parsed incorrectly. This is not acceptable in aerospace systems, where one mistake in the voltage values, or one mistake in the calculation of the mass of the spacecraft can lead to a total failure of the mission. Therefore we propose a different approach, instead of extracting information from the entire requirement in a single step, we extract the information in multiple steps depending on its grammatical structure. As it can be seen in Section 2.1, an atomic and unconditional requirement has three parts, *Subject*, *Action* and *Value*. If the information given in all the three parts can be reliably extracted into a machine understanble format, the requirement can be automatically validated. Our methodology for extracting each part of that information as follows;

Extracting the Subject: By extracting the *Subject*, we aim to find out which artifacts are responsible for satisfying the requirement. However, when the requirement is traced to the model artifacts, this information is already present. From the traceability links, it can be automatically determined that, which artifact should satisfy which requirement.

Extracting the Value: For extracting the *Value* information, we propose using the requirement attributes. In Section 2.1, the requirement attributes are explained. These attributes are in general used to give extra information about the requirement, however, they do not give any information about the semantic of the requirement. Many tools, as well as standards like ReqIF allow creation of custom attributes for the requirements. Consider our example from Table 3.1, the *Value* given here is the numerical information about the voltage. This numerical information can be presented in a requirement attribute. Since it is a physical quantity, the number itself is meaningless with-

CUS-EPS-16.1	assembly	platform	applicable
All units onboard the S2TEP satellite connected to the unregulated battery bus shall operate nominally with 8.0 V as minimum voltage interface (measured at the unit power interface): The above value is the mean voltage value, and is excluding noise, ripple and voltage spikes).			
Verified by test			
Payload-relevant (the same applies to payloads)			
CUS-EPS-16.2	assembly	platform	applicable
All units onboard the S2TEP satellite connected to the unregulated battery bus shall operate nominally with 10.0 V as nominal voltage interface (measured at the unit power interface): The above value is the mean voltage value, and is excluding noise, ripple and voltage spikes).			
Verified by test			
Payload-relevant (the same applies to payloads)			
CUS-EPS-16.3	assembly	platform	applicable
All units onboard the S2TEP satellite connected to the unregulated battery bus shall operate nominally with 12.0 V as maximum voltage interface (measured at the unit power interface): The above value is the mean voltage value, and is excluding noise, ripple and voltage spikes).			
Verified by test			
Payload-relevant (the same applies to payloads)			

Table 3.2: Dividing the Requirement from Table 3.1 to multiple requirements

out the unit. Therefore, the unit can be presented in another requirement attribute. The resulting requirements would look like Table 3.3. However, *Value* part of a requirement does not necessarily contain a numerical value. Consider another example from the S2TEP project defined in Table 3.4. On that requirement, the *Value* part of the requirement is the listed enumeration. Therefore, these information can be represented in another attribute. A possible re-elicitation can be seen in Table 3.5. When written in a separate attribute, it becomes easier to detect the relevant information about the *Value* part of the requirement and it can be directly parsed by the computer. With the extracted information, numerical comparisons and string matching can be performed depending on the type of the information.

Extracting the Action The next information to extract from the requirement is *Action*. We have

ID	Description	Quantifiable Value	Unit
CUS-EPS-16	All units onboard the S2TEP satellite connected to the unregulated battery bus shall operate nominally with full performance over the following DC voltage interface as minimum voltage:	8.0	Volt
CUS-EPS-17	All units onboard the S2TEP satellite connected to the unregulated battery bus shall operate nominally with full performance over the following DC voltage interface as nominal voltage:	10.0	Volt
CUS-EPS-18	All units onboard the S2TEP satellite connected to the unregulated battery bus shall operate nominally with full performance over the following DC voltage interface as maximum voltage:	12.2	Volt

Table 3.3: Rewriting of the Requirement from Table 3.1 using attributes

suggested to present the *Value* part of the requirement in a requirement attribute, so that it can be reliably parsed by a computer. However, the same strategy would not work if the *Action* part is also presented in another requirement attribute. The reason is that, the information in this part is presented with natural language. Consider the following requirements;

- The mass of the component X should be less than 14 grams.
- The mass of the component X should not exceed 14 grams.
- The mass of the component X should be at most 14 grams.
- The component X should not be heavier than 14 grams.
- The component X should not weigh more than 14 grams.
- The component X should weigh less than 14 grams.

All of the requirements above are semantically same, however their syntax is different. In order to extract the information given in the *Action* part of the requirement, we introduce the validation engines. Validation engines are responsible for performing two tasks, giving the semantical meaning to the *Action*, and performing the validation.

3.3 Validation Engines

In our methodology, the trace elements have a validation engine. Validation engines gather information from the requirement and the model artifact through the trace element as depicted in Figure 3.3. It is possible to classify the validation engines regarding the requirement attributes they require to do the necessary computations. We explain some of the possible engines regarding the attributes we have defined in Section 3.2, however it is possible to define new attributes in the requirements and define new validation engines regarding those attributes. We explain some of the simplest validation engines which are *Inspection*, *Numerical Validation Engines* and *Enumeration Validation Engines*.

3.3.1 Inspection

In order to trace the requirements which cannot be validated automatically, we define a validation engine called *Inspection*. *Inspection* does not provide automatic validation. Instead, the validation is manually done by the user. The *Inspection* engine is used for the requirements which are not obeying the criteria we have defined, or for the requirements which are not providing any information which is suitable for automatic validation. As an example, many non-functional requirements are not providing information to validate automatically. Instead of automatically validating a requirement, the engine can provide a notification to the user when a change in the requirement occurs. With the help of the notification, the user can inspect the artifacts which may be affected by the change, and then the user can manually validate the requirement by reviewing the traced artifacts. *Inspection* engine is not bidirectional, only provides notifications if the requirement is changed. Providing some sort of notification on every change on the model artifacts would force the engineer to review the artifact every time to see if it still satisfies the requirement. However, during the modeling, artifacts change frequently and this review on every change would be time consuming.

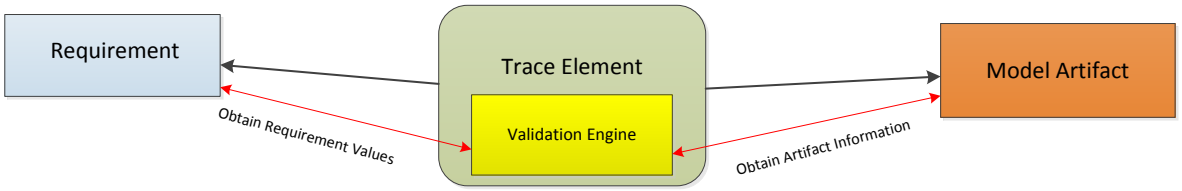
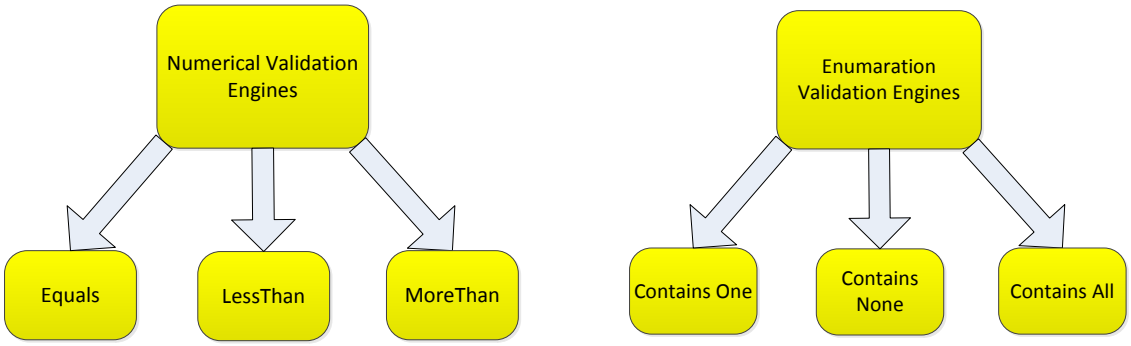


Figure 3.3: Structure of Validation Engines



(a) Numerical Validation Engines

(b) Enumeration Validation Engines

Figure 3.4: Validation engines perform different operations

3.3.2 Numerical Validation Engines

Numerical Validation Engines deal with numerical comparison. They receive a numerical value and the unit of the value from the requirement, and the same attributes from the model artifact. Then they perform the necessary operations. The simplest validation can be to check the equality. Figure 3.4a depicts the structure of numerical validation engines. One important step in order to validate requirements with numerical value is the unit conversion.

Unit Conversion

Numerical Validation engines should consider units of the quantifiable values. Otherwise, semantically equal values such as 1 kg and 1000 grams can be understood as different values. In order to avoid this problem, the methodology should also include a unit conversion logic. This conversion can be done in the three parts. The first option is to limit the requirement units and define a base unit for each quantity, such as "all of the values regarding the mass should be written in grams". However, limiting the requirements would have two problems. First one is, it limits the requirements sharing. With that approach, when a user receives a requirement document from another project which contains all the mass values in "kilogram" units, then the user manually has to convert the "kilogram" units to grams. Second problem is that, even if the requirement units are limited to base units, to avoid the conversion, the data model should also only have values in base units. However, this may not the case in many implementations. Different data models may have different types of units for modeling purposes.

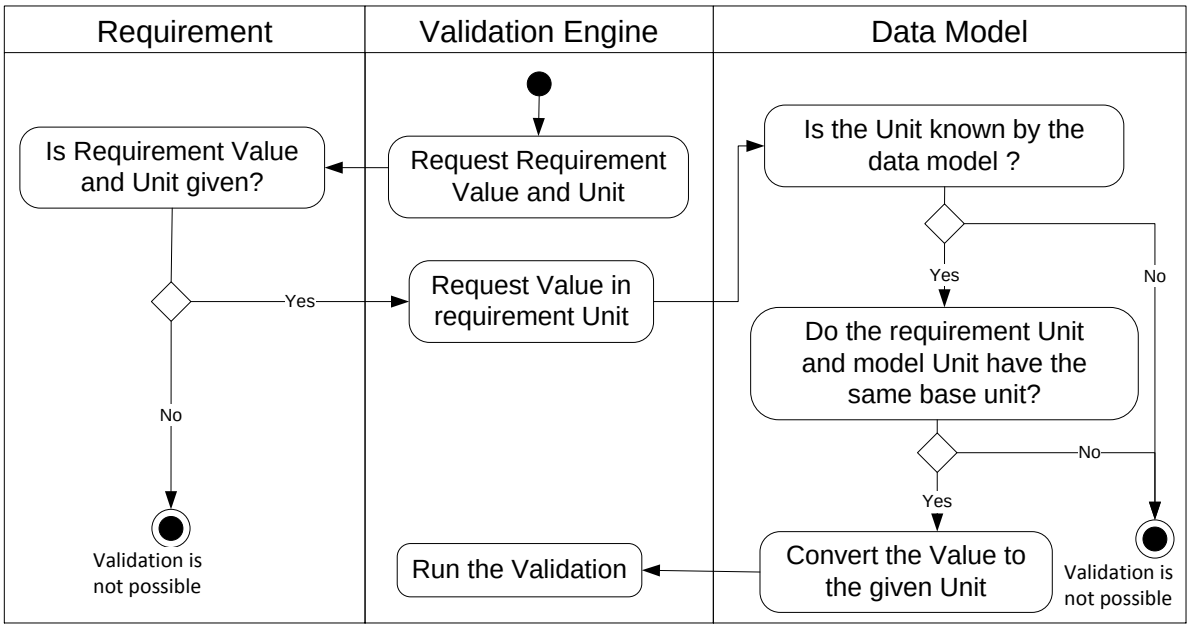


Figure 3.5: Unit Conversion

The second option would be to handle the unit conversion in the validation engines. With that approach, the validation engine would simply request the value and the unit both from the requirement and the data model. Then it converts one unit to the other one and performs the necessary validation operations. However, this approach has also some drawbacks. Since our methodology does not depend on one single model, and applicable to any kind of data models, the validation engine should be aware of all the existing units in the world. This restriction on the validation engine is hard to implement. Moreover, if the validation fails because of the units, it would be hard to detect, if the units are not actually from the same kind, or engine failed due to some implementation mistakes.

The third option is to make the data model responsible for the unit conversion. We have chosen this approach in our methodology. With this approach, we are able to isolate the validation part from the unit conversion logic. Moreover, the users are able to extend their data model if they would like to use new units. Validation engines will run, as long as the data model is aware of the units defined in the requirements. Otherwise, validation engine will not run since the necessary input is incomplete.

Unit conversion can fail in three different ways. Figure 3.5 explains the general flow of unit conversion. First one is; if the unit is not defined in the requirement. In that case, although the requirement is providing a quantifiable value, without the unit, that value is meaningless. Validation will not be possible. Second one is; if the unit of the model artifact and the unit of the requirement are not from same kind. As an example, if a requirement with unit "Kg" is traced to an artifact which has the unit "Volt". In this case, unit conversion will not be possible and therefore validation will not be possible. Third one is, if the model does not know the unit. As an example, if a requirement with unit "Lb" is traced to an artifact with a unit "Kg". In that case both of the units are from the same kind, used to measure the mass. However, if the data model does not know what an "Lb" is,

CUS-OBC-46	assembly	platform	applicable
All data interfaces connected to the OBC shall be selected from the following types of interfaces: -RS422 UART -SpaceWire			
Limiting the number of interfaces types reduces the complexity.			
Verified by review of design			
Payload-relevant (the same applies to payloads)			

Table 3.4: Example Requirement From S2TEP Project with enumeration

Id	Description	Enum Values
CUS-OBC-46	All data interfaces connected to the OBC shall be selected from the following types of interfaces:	RS422 UART, SPACEWIRE

Table 3.5: Rewritten Requirement From S2TEP Project with enumeration

then the conversion is not possible and therefore validation is also not possible.

3.3.3 Enumeration Validation Engines

Another information to present in the *Value* part of a requirement is enumeration such as in the example requirement shown in Table 3.5. In that particular example, the model artifact should contain one of the items defined in the enumeration. However, another similar requirement may request that the model artifact should contain all of the items or a requirements can also prohibit the usage of the listed items. Depending on the need, different enumeration engines can be defined. Enumeration engines check if the model artifact contains all of the enumerated items, or contains some of them, or contains none of them. Some possible variations can be seen in Figure 3.4b.

3.3.4 Further Capabilities of the Validation Engines

The example engines we explained in Figure 3.3, directly compare the information given in the requirement and the information present in the data model artifact. Besides the simple ones, validation engines can have more complex character. The requirements may give information which may not directly exist in the data model. Validation engines can also derive information from the data model. Consider the following requirement;

- The star tracker should be daily active for at least 8 hours.

The requirement is related with the star tracker, but the activity time of the star tracker strictly depends to the supplied power. In order to derive this information, the requirement should be traced both to the star tracker and the component which supplies power, such as a power control and distribution unit(pcdu). Moreover, even with any other methodology, this requirement should be traced to both the star tracker and pcdu, since a possible change on this requirement can affect both the design of the star tracker or the design of the pcdu. In order to obtain information from both of the artifacts, the requirement should be traced as depicted in Figure 3.6.

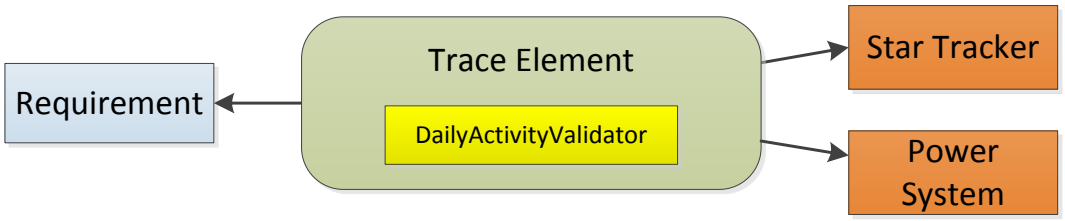


Figure 3.6: Validation engine derives information

Validating requirements with Multiple subjects

Although we have limited the automatic validation to atomic requirements, requirements with multiple subjects can also be automatically validated. One example is the requirements with finite subjects. Consider the following requirement;

- Component X and Y should weigh less than 400 grams.

This requirement can be validated by either tracing it through the model objects through a single trace element, or creating two different trace elements. In the first case, the validation engine should perform the validation operation for each of the artifacts, whereas in the second case, the validation engines of the each trace element should perform the validation for the traced artifact. However, after the requirements we have observed, this finite number of subjects is not common, instead, many of the requirements are about a set of similar components in the model.

Validating the Requirements with System Wide Subjects

After analyzing the space mission requirements, we have observed that, many requirements have system wide subjects. Consider the following example;

- Connectors shall be made of non-magnetic materials with a residual magnetism level below 200 Gamma (200 nT).

As with the requirements with multiple subjects, the requirement can be traced to the connectors with a single trace element, or different trace elements per connector. However, with that approach, the validation of the requirement would be incomplete. The requirement is related with all of the connectors in the model. When new connectors are added to model, they also need to be validated. If any of the newly added connectors are not satisfying this requirement, the engineers should be notified. However, with the tracing approaches we have explained so far this is not possible. In order to handle this situation, we propose using the hierarchy of the data model. In the previous traceability approaches, we have considered data model artifacts as standalone artifacts, and obtained information directly from them. However some data models are hierarchical. In spacecraft modeling, the spacecraft has different components, and those component have different sub-components. In order to reliably validate the requirement, this requirement should be traced to the root of the model. Then the validation engine should perform the validation operation for each of the connectors. This approach would ensure that, if new connectors are added to the data model, they will also be subject to this requirement. An example tracing can be seen in Figure 3.7. However in this case, since we are not tracing the requirement directly to the artifact, validation engine also needs

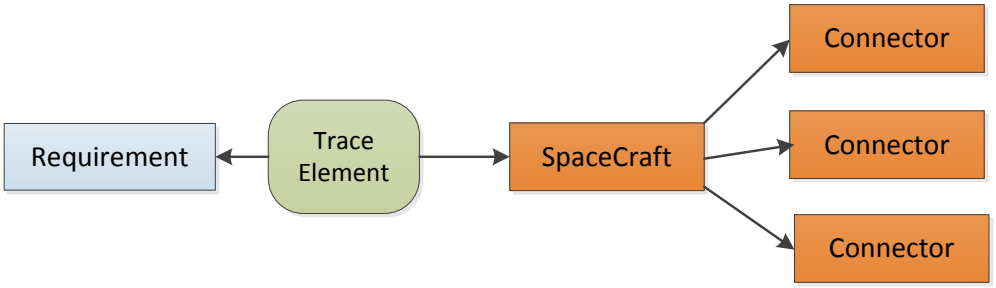


Figure 3.7: The validation of the connectors

to know what type of elements it will validate. Moreover, which attribute should be validated is not easy to detect since it is not directly traced to an attribute of a single artifact. This approach would require different validation engines for different types of artifacts and different types of attributes.

Validating the requirements with Object Dependent Subjects

Similar with the system wide requirements, many requirements are posing a limitation on the components depending on their connection with other components. Consider the requirement we have shown in Table 3.3. The requirement contains multiple subjects. Similarly with the previous example, tracing this requirement to the individual components would not validate the requirement since new components can be connected during the modeling phase. Moreover, this requirement is relevant on a component, as long as the component is connected to the unregulated battery bus. If at some point, engineers decide to connect the existing component to regulated bus, this requirement would not be relevant for the component anymore. In order to reliably validate the requirement, this requirement should be traced to the object. Using the information contained in the data model, each connected component can be individually validated. The validation engines responsible for validating such requirements should carry the object information as well as the attribute which needs to be validated. In this example, the requirement should be traced to the unregulated battery bus, and the attribute is the voltage value.

Validating the Requirements with Lower Level Elements in Hierarchy

If there is a requirement about a component, it should not be necessarily directly traced to the component. Depending on the granularity of the model, existing requirements may need to be traced to the lower level artifacts. Consider the following requirement;

- The on-board computer shall provide a non-volatile data storage capacity equal to or greater than 260 Mbyte.

If the data model treats the on-board computer as a single unit, this requirement should be traced to the on-board computer. However, if the parts of the on-board computer is also modeled, this requirement should be traced to the hard drive of the on-board computer. Tracing this requirement directly to the on-board computer would require the review of the entire on-board computer, when a change on the requirement occurs. However, tracing the hard drive would ensure that, if the requirement changes, only changing the design of the hard drive would be sufficient.

ID	Description	Quantifiable Value	Unit
CUS-EPS-16	All units onboard the S2TEP satellite connected to the unregulated battery bus shall operate nominally with full performance over the following DC voltage interface:		
CUS-EPS-16.1	minimum voltage	8.0	Volt
CUS-EPS-16.2	nominal voltage:	10.0	Volt
CUS-EPS-16.3	maximum voltage:	12.2	Volt

Table 3.6: Rewriting of the Requirement from Table 3.1 with subrequirements

Detecting Requirement Inconsistencies

In Section 2.1, it is explained that, a good requirements document should not contain conflicting requirements. However, this may not be the case in the practice. Different requirements are generated by different people, and there is always a room for human mistake. With our automatic validation, these requirement inconsistencies can be captured in the modeling phase. Consider a scenario where there are some general requirements affecting all of the components of the model. When generating system and sub system requirements, if a requirement which is not consistent with the general requirements are added, this can be detected by the automatic validation mechanism. If such a case occurs, engineers would receive a violation notification on a model artifact, and the violation would not be fixed by modifying the existing artifact. In such a case, it would be possible to deduce that the existing requirements are conflicting with each other.

3.4 Tracing Without Automatic Validation

In the previous section, we have explained our methodology for automatic validation. We have excluded non atomic and conditional requirements for the automatic validation. In this section, we explain the challenges of non atomic and conditional requirements. Moreover, requirements which cannot be automatically validated should also be traced to relevant model artifacts. The procedure we elaborate in Subsection 3.4.3.

3.4.1 Challenges of Non Atomic Requirements

In our methodology, we required requirements to be atomic to automatically validate them. One important reason is that *Value* part of the requirement is obtained through requirement attributes. Non-atomic requirements contain multiple *Values*. These multiple values can be expressed in multiple attributes, however they do also contain multiple *Actions*. We use one validation engine in order to give semantics to the *Action*. In order to provide the matching between the particular *Action* and a particular *Value*, the engine needs to specify which *Value* it needs. This can be achievable, by providing such engines. However, instead of converting the requirement into a structure with multiple attributes, the requirement can also be decomposed into multiple requirements. Since both of the modifications require manual work, in our methodology we decompose the requirements with multiple information. If for some reason, it is not allowed to increase the number of the requirements, or to add new requirements, the original requirement can be divided in subrequirements as in Table 3.4.1.

3.4.2 Challenges of Conditional Requirements

We have left out the conditional requirements for the automatic validation and focused on non conditional requirements. In order to validate a conditional requirement, first the condition needs to be checked, if the condition is true, than the requirement should be validated. However, in order to check the condition, the information in the condition should be also extracted. We extract the subject information through the trace elements. However, in order to extract the subject of the condition, the requirement should be traced to the model artifact which is related with the condition. Consider the following requirement;

- The notification system shall send error messages if the water tank has less than 10 liters of water.

This requirement should only be traced to the notification system. Although it contains information about the water tank, it should not be traced to the water tank. The reason is that, possible changes on that requirement are not affecting the design of the water tank. Since the requirement is not traced to the water tank, it cannot be computed if the condition is satisfied or not. Because of this shortcoming, conditional requirements are left out of the scope for automatic validation.

3.4.3 Tracing with Inspection Engine

Automatic validation of the requirements is not possible in several cases, such as with non atomic requirements, however, those requirements should also be traced to the relevant model artifacts in order to identify the impact of a requirement change on the model artifacts. Non atomic requirements can contain multiple *Actions* and *Values* , multiple *Subjects*, or they can be a combination of two atomic requirements. We create one trace element per *Action*, and trace the components related with that *Action* with the trace element. Consider the example requirement;

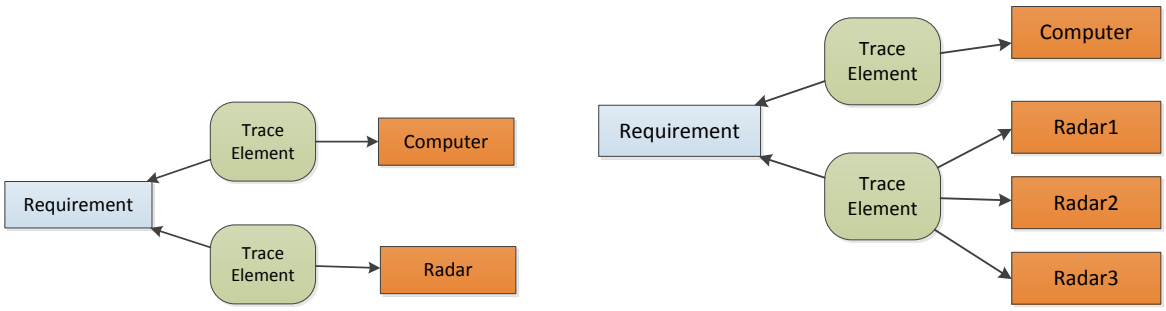
- The system shall be able to detect the objects within 200 m and send notifications in 3 seconds.

Assuming that the system has a radar to detect objects, and a computer to send notifications, the requirement should be traced as depicted in Figure 3.8a. Moreover, if the object detection is conducted by different elements, such as three different radars combined the components are traced as depicted in Figure 3.8b.

If the first part of the requirement changes, only reviewing the elements which are traced to the requirement through the first trace element is enough, on the other hand, if the second part of the requirement changes, only the artifacts traced with second trace element should be reviewed. This can be achieved with assigning proper description and name to the trace elements, this difference would allow the reviewer to avoid redundant reviews.

3.5 Managing the Requirements and Trace Elements

In the previous sections, we have covered how to modify the requirements to make them suitable for automatic validation as well as our traceability model. In this section, we explain how to manage the requirements document, as well the trace elements during a project.



(a) Detection is done with a single radar

(b) Detection is done with multiple radars

Figure 3.8: Tracing model artifact with semantics

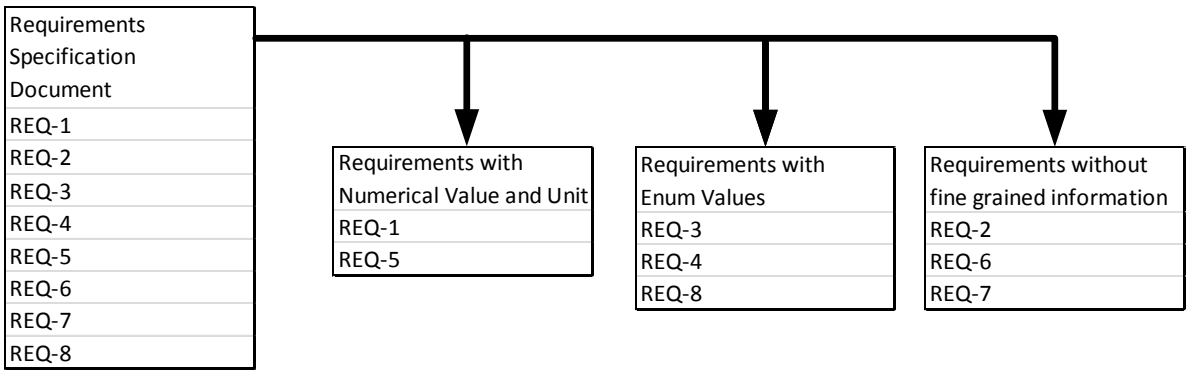


Figure 3.9: Splitting the requirement file to multiple specifications

3.5.1 Managing the Requirements Specification Document

A project can contain multiple requirement specification documents. These can be hierachical such as high level system design requirements and low level system design requirements. For space mission requirements, as explained in subsubsection 2.3, different sub-system requirement documents and different equipment requirement documents may exist. These documents are written by different domain experts. Typically, requirements belonging to the same document carry the same requirement attributes. With our methodology, we provide the *Value* information in requirement attributes. In Table 3.3 and Table 3.5, we have only demonstrated value, unit, and enumeration as requirement attributes. However, depending on the information presented in the requirements, this list can be extended and a large number of different attributes may need to be defined. Adding attributes to the requirements, which will not be used, is a redundant presentation. To overcome this issue, the requirements document can be split into different specifications[27] regarding ReqIF standard. In ReqIF standard, requirements document can have multiple *Specifications* and each *Specification* can have a different *SpecType*[27]. Each *SpecType* can have different attributes. Considering the previous examples, the initial requirements specification document can be reformatted as seen in Figure 3.9. In this example, the resulting document has three *Specifications*.

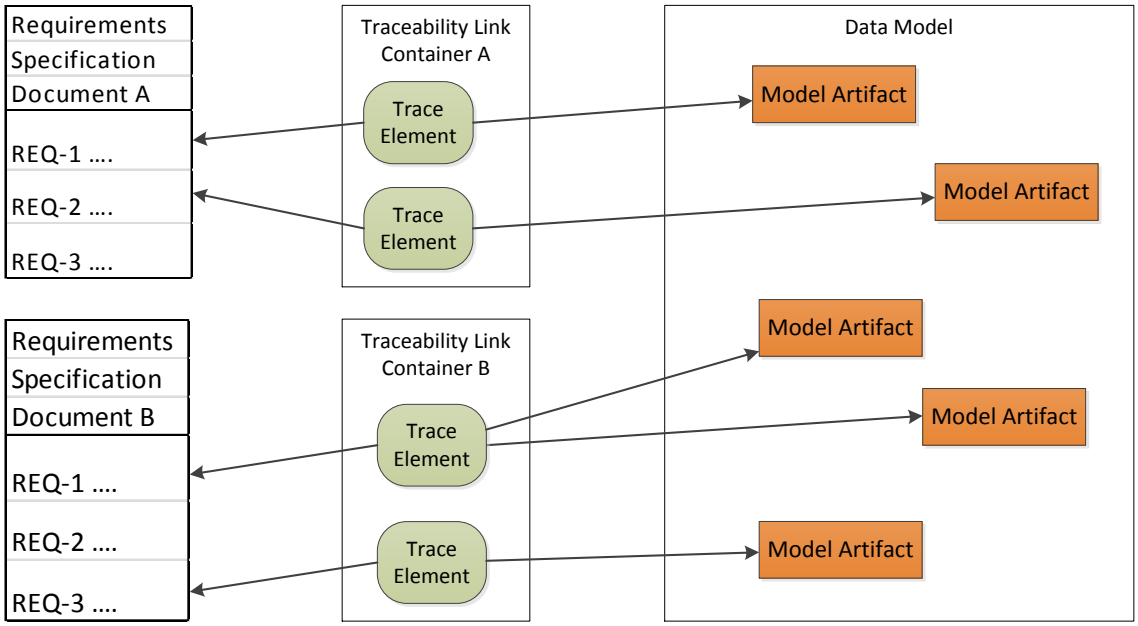


Figure 3.10: General architecture of Traceability Link Containers

3.5.2 Storing the Trace Elements

In our methodology, we group the trace elements depending on their requirements origin. Trace elements referring to the requirements which are in the same requirement specification document should be grouped together. For that purpose, we define traceability link containers. There is exactly one traceability link container for each requirement specification document. Traceability link containers should contain individual traceability links related with the requirement specification document. The architecture of the traceability link containers are depicted in Figure 3.10.

3.5.3 Managing the Trace Elements

A change on a requirement does not necessarily occur on the *Value* part. Depending on the change, existing trace elements of the requirements need to be manually updated. We define four different possible changes on a requirement. These are;

- Delete; The requirement is deleted.
- Merge; Multiple requirements are merged into a single requirement.
- Decompose; The requirement is decomposed into two or more requirements.
- Modify; The contents of the requirement is modified.

The action needs to be taken on each possibility is as follows;

Delete: If the requirement is deleted, all of the trace elements of that requirement should also be deleted. This action also can be automatically performed.

Merge: If requirement A is merged with requirement B; the source of the the trace elements should be updated. This part can be handled automatically. However, that may create redundant traceability information. Consider two requirements traced to the same model artifact without automatic validation, merging those requirements would create two trace elements with the same source and same target. In that case one of the trace elements can be manually deleted.

Decompose: The action after a decompose cannot be automated because, if the requirement is traced to multiple model elements, which which model element is associated with the which part of the requirement is not easy to detect. Moreover, it is not a simple redistribution of the trace elements, some trace elements may required to be duplicated. Consider the following requirement;

- The on-board computer shall provide the ability to execute software applications and data storage.

After the requirements is decomposed in to two atomic requirements,

- The on-board computer shall provide the ability to execute software applications.
- The on-board computer shall provide the ability to store data.

If the requirement is directly traced to the on-board computer with a single trace element, after decomposing the requirement, the trace element should be copied so that both of the resulting requirements are traced to the on-board computer. However, if the requirement is traced to multiple sub-equipment, such as the hard drive and cpu of the on-board computer, the trace elements should be distributed over the resulting requirements after the decompose.

Modify: Action to be taken depends on the changed part of the requirement and the engine of the trace elements. Regardless of the engine, if the **Subject** part of the requirement is changed, the target of the trace element needs to be manually updated. Consider the following change on the requirement,

- **Original Requirement** Component X should be supplied with 8 Volts
- **Changed Requirement** Component Y should be supplied with 8 Volts

In such a case, the source of the trace element should be manually updated to *Component Y*.If the change is not on the **Subject** part of the requirement, for the requirements which are traced with *Inspection Engine*, the traced artifacts should be manually reviewed. For the requirements which are traced with an automatic validation engine, if the **Action** part changes, the validation engine of the trace element needs to be updated. As an example;

- **Original Requirement** The mass of component X should be less than 50 Grams.
- **Changed Requirement** The mass of component X should be more than 50 Grams.

Assuming that the requirement was initially traced with *LessThanValidator*, the engine of the trace element should be change to *GreaterThanValidator*.

If the *Value* part of the requirement changes, validation engine automatically validates the requirement as depicted in Figure 3.11. If the traced artifact is not satisfying the new *Value*, validation engine provides necessary notification. Manual review of the artifacts are not needed.

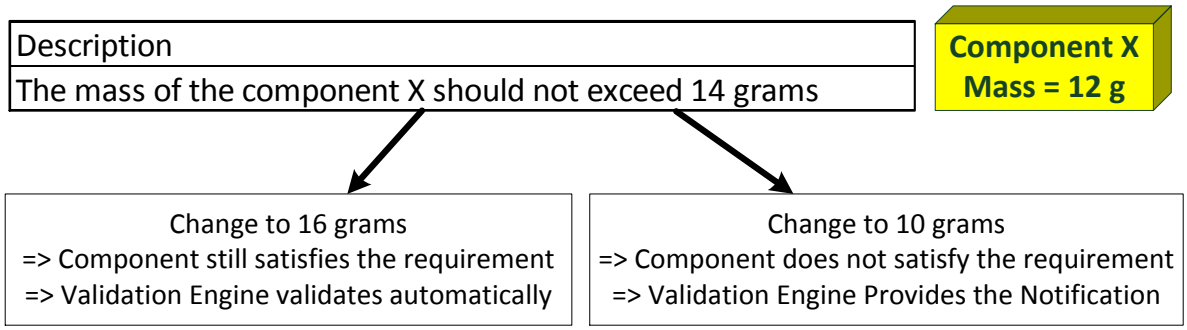


Figure 3.11: Some requirement changes do not affect the model element

3.6 Structured Requirements

In the previous sections, we have discussed how to trace the requirements to the model artifacts and how to automatically validate them. However, we did not limit the language of the requirements. Any kind of requirement with natural language is suitable. This usage of natural language provides couple of advantages. A functionality of the system can be explained in detail. The ideas and design decisions of the system can be precisely expressed with natural language. However, this type of requirements given in natural language has two drawbacks for the automatic validation. First one is, the requirements are manually traced to the model artifacts. Therefore, the validation engine of the trace element is chosen manually. This manual selection is prone to errors. If a false engine is selected, which does not reflect the semantics of the requirement, some false violations can occur. Moreover, actual violations cannot be caught with an incorrect engine which may have devastating impacts on the mission. Second one is, if a change occurs on the *Action* part of the requirement, the validation engine of the requirement needs to be manually updated. This task have the same risks, since there is a possibility that engineers may forget to update it. To address those issues, structured requirements can be used. In this section, we analyze the usage of structured requirements for the automatic validation. We propose using boilerplates with restricted language in mission requirements. We target the requirements which can be automatically validated, since for the other requirements, restricting the natural language would not provide any benefit for the automatic validation. Besides restricting the natural language usage, these patterns can enforce requirements to be atomic, containing one *Action* per requirement.

3.6.1 Templates for the Requirements

In Section 4.3, we gave details on the capabilities of the validation engines, and the requirement types which can be automatically validated. We present the respected structure for each case.

Atomic Requirements with Single Subject

Directly parsing the atomic requirements to extract information is not possible, since the same *Action* could be presented with different linguistic structures. These can be difficult to be understood by the computer. For them, we limit the vocabulary and forbid the negative statements. For the requirements with numerical information, we propose the pattern in Figure 3.12. With such a

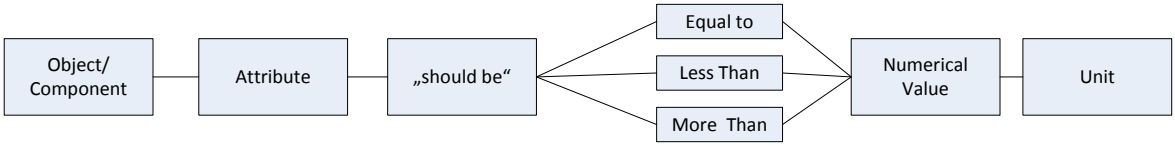


Figure 3.12: Template for requirements which contain numerical information

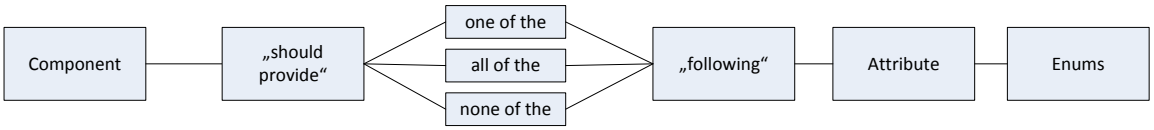


Figure 3.13: Template for requirements which contain enumeration

structure, the validation engines can directly parse the requirement. In order to make the *Action* part of the requirement understandable by the engine, we extend the validation engines. Each engine can have *Action* encoded within them. As an example, for the *EqualsValidator* we can encode the phrase *"equals to"* in the engine. This phrase would be the simplest explanation of which operation is performed by the validation engine. With that approach, the relevant validation engine for the requirement can be automatically detected with a simple text parsing on the requirement. Similarly for the requirements which contain enumeration, we propose the pattern in Figure 3.13.

System Wide Requirements

As we have seen in Section 3.2 some requirements are posing restrictions on all of the elements with the same type, this system wide restriction can be expressed in many different forms with natural language such as

- Connectors shall be
- All connectors shall be
- Every connector shall be

Complete validation of these types of requirements can be achieved by tracing them to the root of the system. We propose the boilerplate in Figure 3.14 to express these type of requirements. However, besides the semantics about the *Action* part of requirement, the validation engines for these type of requirements should also contain the type of the artifact and the attribute to perform the operation on. Similar with determining the *Action*, a simple string parsing would be enough to determine the type of the artifact and the attribute, so that the validation engine can be automatically assigned. Besides the advantage of automatic selection of the validation engine, when these system wide requirements given in structured format, several extra safety checks can be applied. One example is, it can be disallowed to trace these kinds of requirements to individual artifacts.

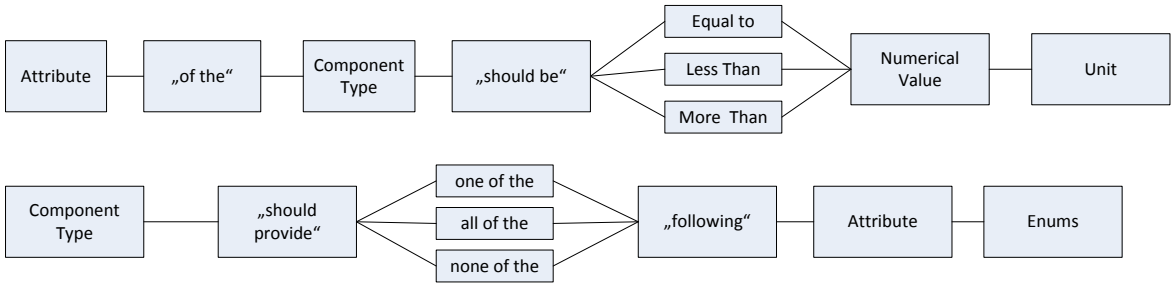


Figure 3.14: Template for system wide requirements

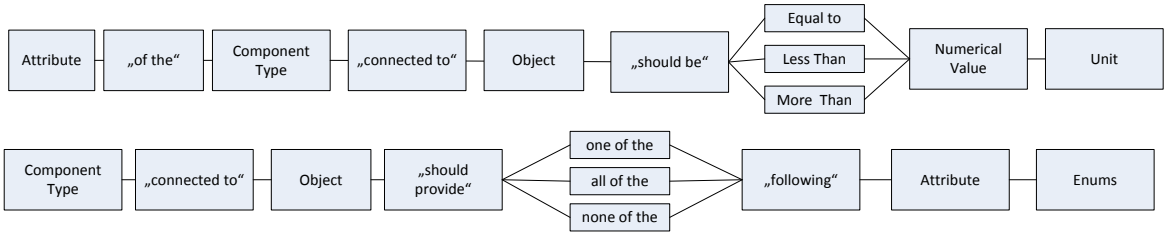


Figure 3.15: Template for object based requirements

Object Dependent Requirements

For the object dependent requirements we have analyzed, we provide the boilerplate in Figure 3.15, when they are given in such a structure, the engine to validate the requirement can be automatically selected. Similar with system wide requirements, these engines should also contain the type of the artifact and the attribute to perform the operation on.

3.6.2 Generating the Structured Requirements

Although the structured requirements are more resilient against the changes for the automatic validation, converting the existing requirements to such structured formats is a manual and time consuming task. Instead, we propose using our traceability methodology to extract such requirements from the existing unstructured requirements which are traced to model artifacts.

Generating the Boilerplate Requirements

If a requirement is traced with automatic validation, the information presented in the requirement can be automatically extracted. The *Subject* of the requirement is the traced artifact and the relevant attribute, the *Action* part of the requirement can be obtained through the validation engine, and the *Value* part of the requirement can be directly copied from the original requirement. Figure 3.16 depicts how this generation can be made. Before sharing the requirements, such structured requirements can be obtained from the requirements which are written in natural language. This generation can be done in most of the cases, however, there are also some limitations. For the following cases, the generated structured requirements cannot be directly used.

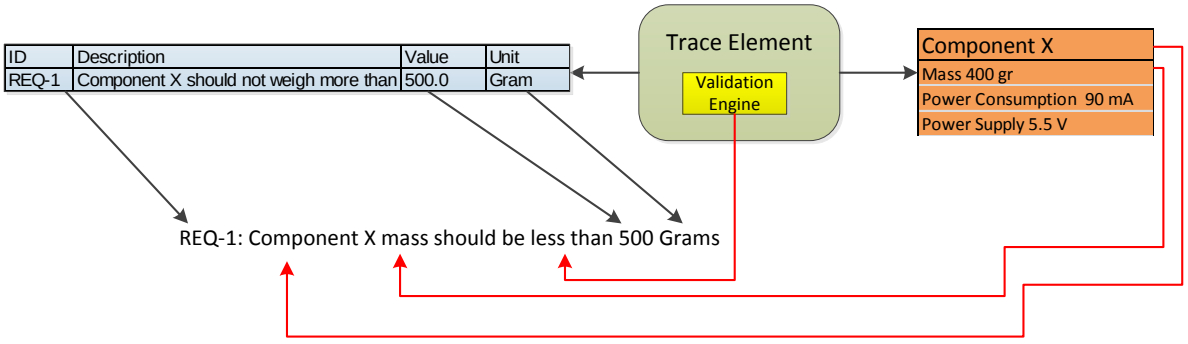


Figure 3.16: Generation of structured requirements

Requirements Traced with Incorrect Engines

Since tracing the requirements to model artifacts is a manual work, inconsistent traceability links may have been created. As an example, if a requirement is traced with an incorrect engine to the model artifacts, generating structured requirements would result with inconsistent requirements. This problem should be manually detected by the engineers.

Requirements Traced to Lower Level Artifacts in Hierarchy

If the sub-system requirements are traced to different equipment, generating boilerplate requirements would change the semantics of the initial requirement. Consider the example in Figure 3.17. The requirement about the power system is traced to different batteries with the different voltages. Generating a boilerplate requirement would result in two requirements about two different batteries. Those requirements would put an unnecessary restriction on the new design, since the initial requirement can be satisfied by a single battery and two different voltage converters. Moreover, same limitation also exists if a requirement about an equipment is traced to a sub-equipment. Consider the same requirement from Section 3.2;

- The on-board computer shall provide a non-volatile data storage capacity equal to or greater than 260 Mbyte.

In this case, the requirement is about a functionality of the equipment, if that functionality is provided by a single sub-equipment, generating a structured requirement would enforce an undesired limit on the design. If the requirement is traced to the hard drive of the on-board computer, generated requirement would contain the hard drive as the subject. This poses an extra limit on the design. This requirement can also be satisfied by using multiple hard drives. In that situation, creating a structured requirement would change the semantic of the initial requirement.

3.6.3 Sharing the Structured Requirements

As we have seen in subsubsection 2.3, sharing and reusing requirements would decrease the cost of a mission. However, in order to find a reusable set, it is necessary to compare the requirements from different missions and find their differences. This cannot be easily done when the similar requirements are given in natural language without any structure. Besides its benefits in automatic validation, requirements with such structure can be reused more easily. If the requirements are

ID	Description	Value	Unit
SRD-EPS-132	The power system shall provide the following voltages:		
SRD-EPS-132.1	regulated	3.3	Volt
SRD-EPS-132.2	regulated	3.5	Volt

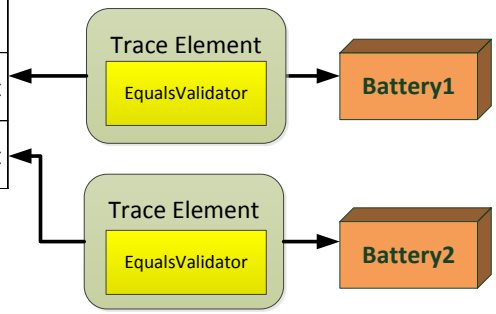


Figure 3.17: The subsystem requirement is traced to equipment

shared with the presented structure, comparing them and finding the similarities and differences would be easier. Furthermore, if one of the existing requirements document selected for the new mission, or a subset of it, structured requirements can be traced to the model artifacts of the new mission more reliably, since the engine for the each requirement can be automatically assigned.

3.7 Modeling on the Basis of Requirements

In the previous section we have described how to trace the requirements to existing model artifacts. In this section, we explain how our methodology can be used to generate model artifacts on the basis of the requirements.

3.7.1 Modeling the Spacecraft

In subsubsection 2.3, the structure of a spacecraft is demonstrated. Usually a component is responsible for each function of the spacecraft, on board computer provides data storage, electrical power system provide power and attitude orbit control system provides orbit control. The requirements about these components define their precise functionality. Hence, the components are modeled in a way to ensure this functionality regarding the requirements. Modeling those components on the basis of the requirements would provide two advantages to the engineers. First, it would be time saving for the engineers to directly generate artifacts on the basis of a requirement, instead of first creating the artifact and then creating a traceability link between the artifact and the requirement. After the generation of the artifact, a trace element can be automatically created with references to the requirement and the newly generated model artifact. Since the validation engine must be set by the user, newly generated trace element should either have the *Inspection* as the validation engine. Second, manual tracing is prone to human errors, where it is possible to trace a requirement to a different artifact. Generating the artifacts on the basis of the requirement would ensure that, the traceability information between the model artifact and the requirement is correct. Figure 3.18 depicts how the sub-systems can be created from the existing requirements in the early stages of modeling. Moreover, requirements document may change during the modeling of the mission. In the later stages of modeling, artifact generation on the basis of requirements can still be useful, since new requirements can be included and they may require new artifacts to satisfy them.

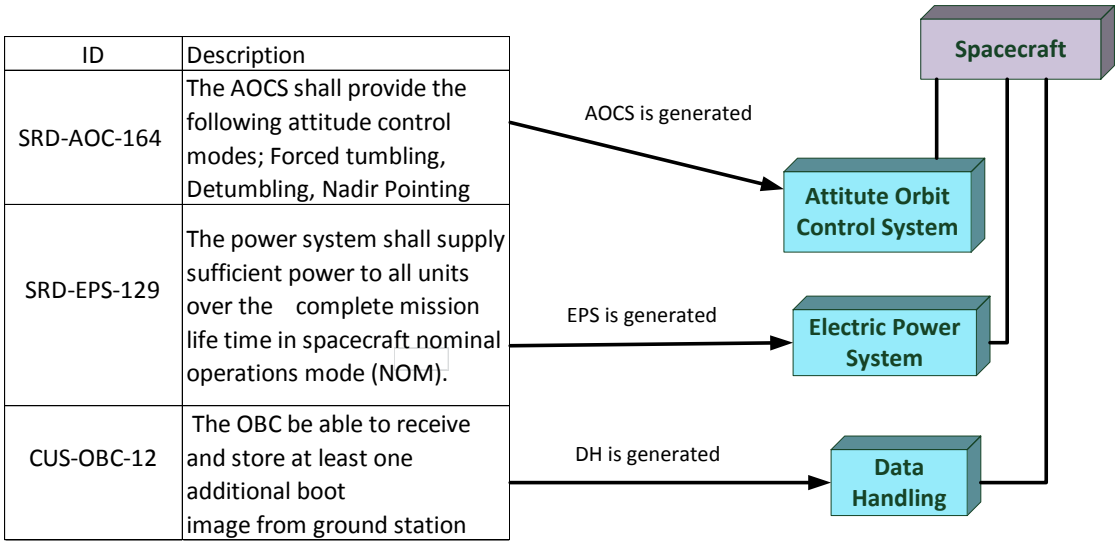


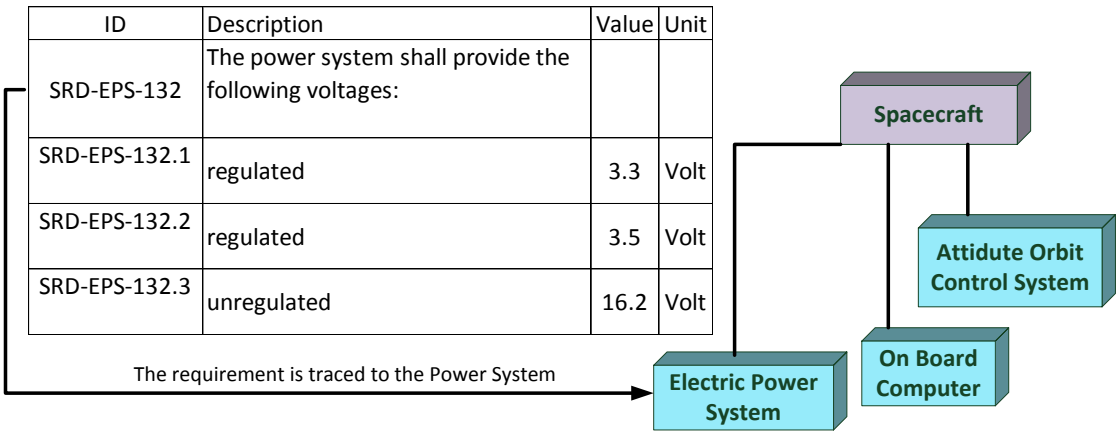
Figure 3.18: First artifacts of the model can be directly created from requirements

3.7.2 Modeling in a Deeper Hierarchy

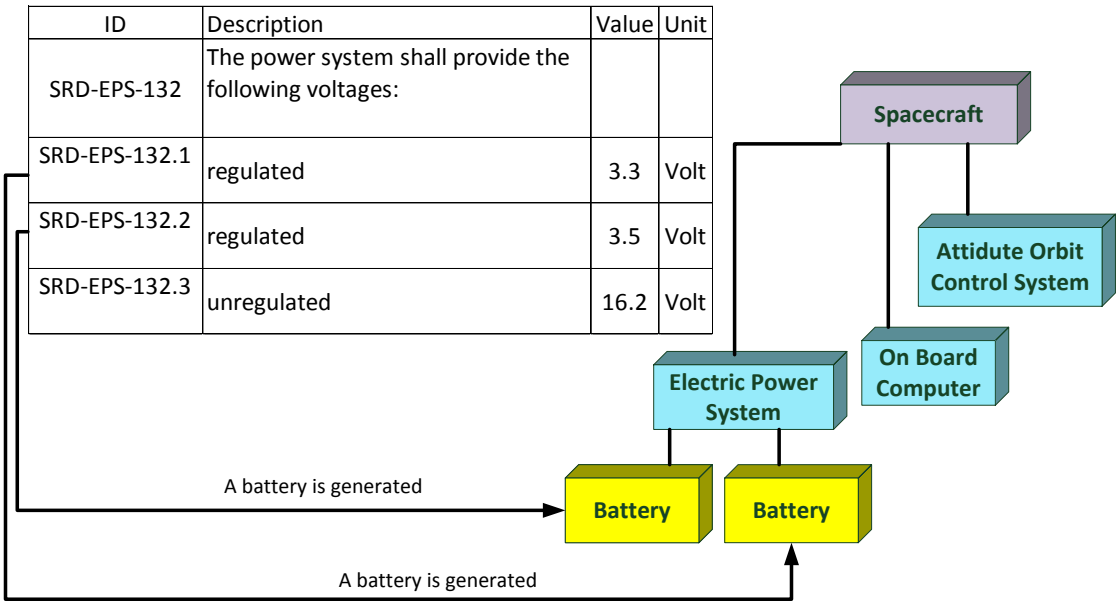
The spacecraft can be modeled in different abstraction levels. A spacecraft can be presented by just modeling the sub-systems or a spacecraft model may contain the respected equipment for each of the sub-systems. If the spacecraft is presented by just modeling the sub-systems, requirements about a sub-system should be traced to the respective sub-system. However, when the engineers start modeling the spacecraft in a deeper hierarchy, requirements traced to a sub-system should be divided and traced to the respected equipment. These equipment can be directly generated from relevant requirements to ensure the completeness of the decomposition. As an example, in Figure 3.19, the requirements related with the voltage values are traced to the power system. However, when the engineers model the equipment of the power system, they can determine which equipment should be created depending on the requirement. In the example, power system needs to provide two different voltage values. The power system can be developed in various ways to achieve this functionality. However, if the engineers decide to use two different batteries for each of the voltage value, then create those batteries can be created on the basis of respected requirements.

3.7.3 Changes on Existing Requirements

In the previous section it is shown that, if the requirement is traced with an automatic validation engine, the changes which violate the design can be automatically detected. However, some of those changes may not be handled by simply modifying the model artifact. Instead, the change may require a new component to be added. Figure 3.20 shows how such a change can be handled using artifact generation. In Figure 3.20 the existing requirement is traced to a battery. That single battery can provide 2 years of lifetime to the satellite. However, if the life expectancy of the satellite increases, one more battery should be added to the satellite. The new battery can be generated from the requirement, with the traceability link included.



(a) Voltage requirements are traced to EPS



(b) Two batteries are generated to provide different Voltage values

Figure 3.19: Modeling in deeper hierarchies

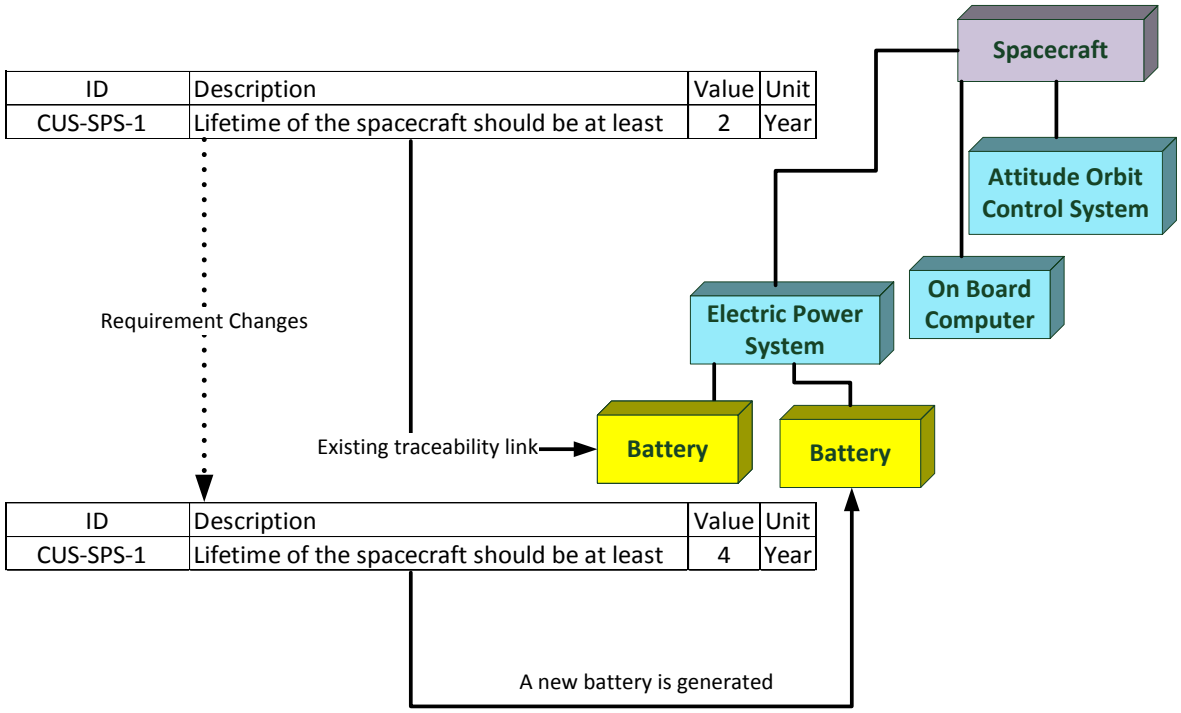


Figure 3.20: A new battery is generated from the changed requirement

3.7.4 Modeling the Budgets

Not only the components of the spacecraft can be generated on the basis of requirements, but also the respected budgets for sub-systems and equipment. In a scenario, where the mission requirements are defined, however, the specific requirements for sub-systems and equipment are not settled, engineers can first work on the dimensions and positions of the components of the spacecraft. In order to guide the further design, engineers can create the budgets for the sub-systems and the equipment on the basis of the requirements. When the engineers start to model the components, they consider the allocated budgets as a limitation for their design. This is another scenario where artifact generation on basis of requirements would save effort. In Figure 3.21, from the mass limitation of the spacecraft, mass budget of the attitude orbit control system is generated. Furthermore, mass budgets for the equipment of the sun-system can also be generated. Furthermore, in the later stages, these budgets can also change. As an example, if the engineers cannot design a component regarding the allocated mass budget, they may require extra budget by making another component lighter. In such a case, necessary notifications can be automatically provided to the engineers.

3.7.5 Extending the Data Model

Proper traceability and automatic validation is only reachable, if the information presented in the requirement also exists in the model. If the information is not modeled, then the requirement cannot be successfully traced. However, modeling tools evolve over time and new functionalities can be introduced. Which means that, the data model can be extended over the time, so that new types of model artifacts can be created and traced for automatic validation or just simple traceability

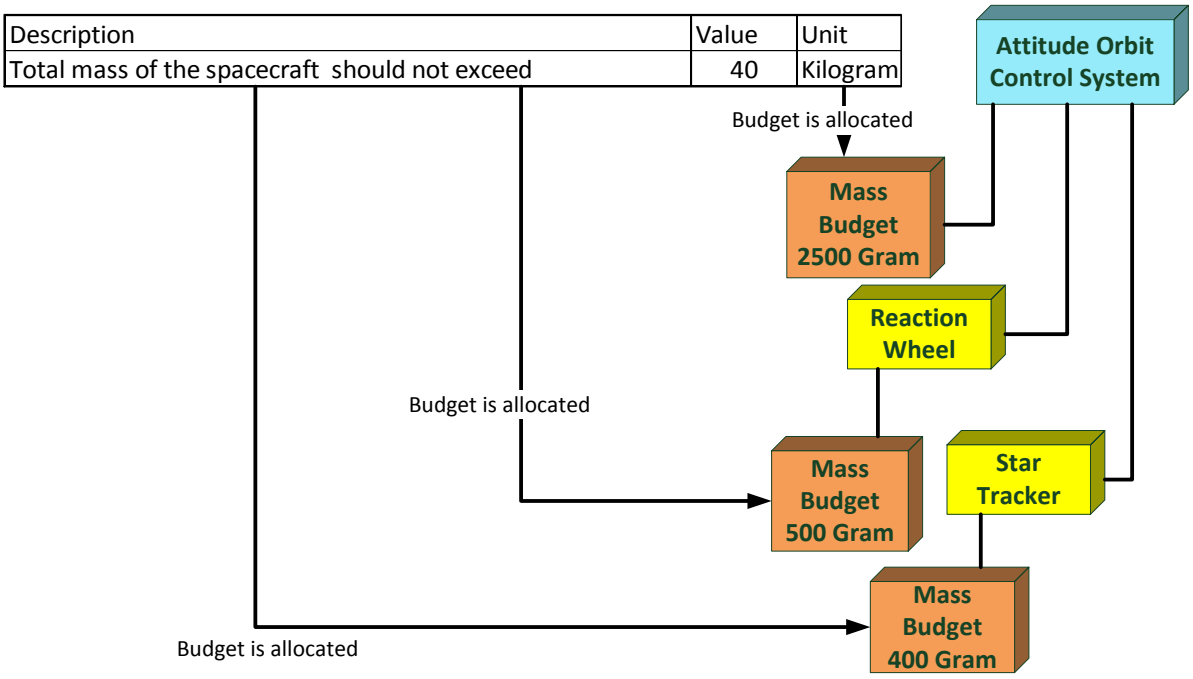


Figure 3.21: Sub-system and equipment budgets are generated

information. In such cases, existing requirements which could not be traced due to lack of model knowledge, can be traced to the newly generated artifacts. In such a situation, these new artifacts can be generated on the basis of the existing requirements.

3.7.6 Generating Artifacts on the Basis of Trace Elements

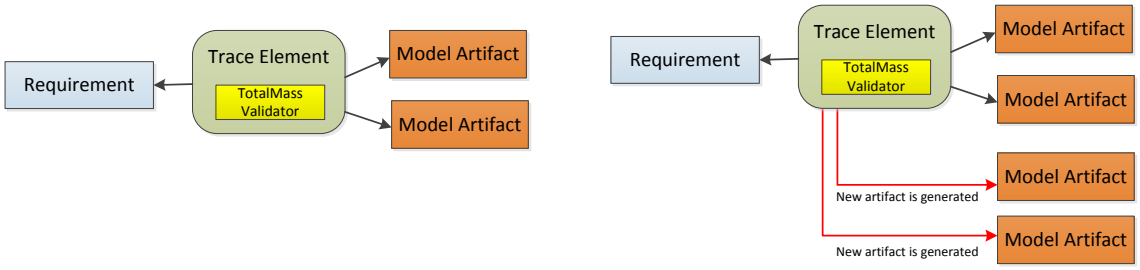
In subsubsection 3.3.4 we have suggested that the system wide requirements should be traced to the root of the model to ensure complete validation. This complete validation can also be guaranteed, if the artifacts related with the system requirements are generated on the basis of a trace element. Consider the following requirement.

- The overall mass of the spacecraft including all payloads and launch adapter should be smaller than 50 kg.

If the requirement is traced to individual artifacts, as depicted in Figure 3.22a, then adding new artifacts would violate the requirement. However, this situation can be avoided if the new artifacts are generated on the basis of the trace element. This procedure depicted in Figure 3.22b. This generation would ensure that, mass of the each artifact is also computed in order to validate the requirement.

3.7.7 Safety checks

As we have elaborated, artifacts can be generated on the basis of requirements on different scenarios. However, even if the generated artifact is directly responsible from the requirement, generating that artifact under a different component would violate the general structure of the spacecraft. Unfortunately, a safety check cannot be inserted in this abstract methodology level, thus, it should be



(a) The requirement is traced to individual artifacts

(b) New artifacts are generated on the basis of the trace element

Component X
Mass 50 gr
Power Consumption 20 mA
Power Supply 3.7 V
Operating Temparature 45 C

Component Y
Mass 60 gr
Power Consumption 30 mA
Power Supply 2.7 V
Operating Temparature 60 C

Component Z
Mass 70 gr
Power Consumption 20 mA
Power Supply 5.7 V

Figure 3.23: The existing components from previous projects

handled by the data model, where hierarchical relations between model artifacts are known. On the other hand, requirements also provide a nested hierarchy, with system requirements to sub-system requirements to equipment requirements. This hierarchy provides the information about the parent component. Therefore, we can assume that, engineers who are working with the requirements have the knowledge to create the artifact under the correct component.

3.8 Requirements-Based Artifact Reuse

In Section 3.6 we have provided requirement templates in order to increase the reusability of the requirements. However, not only requirements can be reused, but also the existing models artifacts can also be shared between engineers and can be reused for the new mission. In this section, we explain the third step of our methodology, reusing the existing model artifacts based on requirements.

As mentioned in the S2TEP case in Section 2.3, parts of the existing design from the previous missions can be reused in the newer missions in order to reduce the cost of the mission. Typically, each mission has its own requirements document. In order to reuse a design from the previous missions, the engineer needs to know which of the new mission requirements are satisfied by the existing design. However, manually finding out which of the requirements are satisfied by a model artifact is time consuming. Furthermore, this manual process is prone to the errors. An engineer can easily misread a requirement, or the engineer can miss a feature of the design. Especially when there are large number of requirements, manually checking the re-usability of the existing artifacts is inefficient. Instead of performing this operation manually, it is possible to use our automatic validation technique to determine reusable model artifacts for the new mission.

ID	Description	Value	Unit
Req-1	Component power consumption should not exceed	40	mA
Req-2	Component mass should not exceed	55	Gram
Req-3	Component should be supplied with	3.7	Volt
Req-4	Operating temperature of the component should not exceed	50	Celcius

Table 3.7: The requirements for the new project

ID	Description	Value	Unit
Req-1	Component power consumption should not exceed	40	mA
Req-2	Component mass should not exceed	55	Gram
Req-3	Component should be supplied with	2.7	Volt
Req-4	Operating temperature of the component should not exceed	50	Celcius

Table 3.8: The requirements which cannot be satisfied completely by the existing design artifacts

Consider the different components which are designed for the previous missions in Figure 3.23. When the requirements document in Table 3.7 is received for the new mission, it can be observed that existing *Component X* satisfies all of the requirements. In this case, engineers can directly reuse the existing design.

However, consider the requirements in Table 3.8, in that case, none of the existing design elements are satisfying the complete set of requirements. Therefore, the requirements can be prioritized and the important ones can be selected. For the remaining ones, it is possible to talk with the stakeholders or the engineers to change the requirements which are not satisfied by the existing design. Moreover, the violation can be compensated on the other artifacts. For instance *Component X* is satisfying every requirement but not REQ-3, *Component Y* is satisfying every requirement but not REQ-1. *Component X* can be reused in the existing design as long as the extra mass can be compensated by designing another component lighter. *Component Y* can be reused with an extra voltage converter.

Besides directly violating a requirement, such as *Component Y* violates the mass requirement, the existing artifact may not contain the information presented in the requirement. As an example, consider REQ-5, operating temperature is not modeled in *Component Z*. If the REQ-2 and REQ-5 are selected to find an artifact, *Component Z* can be considered as a candidate artifact to be used in the new mission. Engineers can decide, if the necessary modifications can be made on the artifact to make it suitable with the new requirements. One important aspect to notice is the compliance of the model artifacts with system wide requirements. The example requirements we have shown are about a single attribute of a single component. However, even if a components satisfies all the requirements, another attribute of the components may not be compatible with the system wide requirements. For such cases, the existing component cannot be directly used. This violation can be detected after adding the component to the new model, as long as the system wide requirement is traced to the root of the model. Alternatively, manual inspections can also be performed by the engineers. Similarly, object dependent requirements also effect the usability of the existing design. When reusing a previous design, engineers should consider the connection between the existing

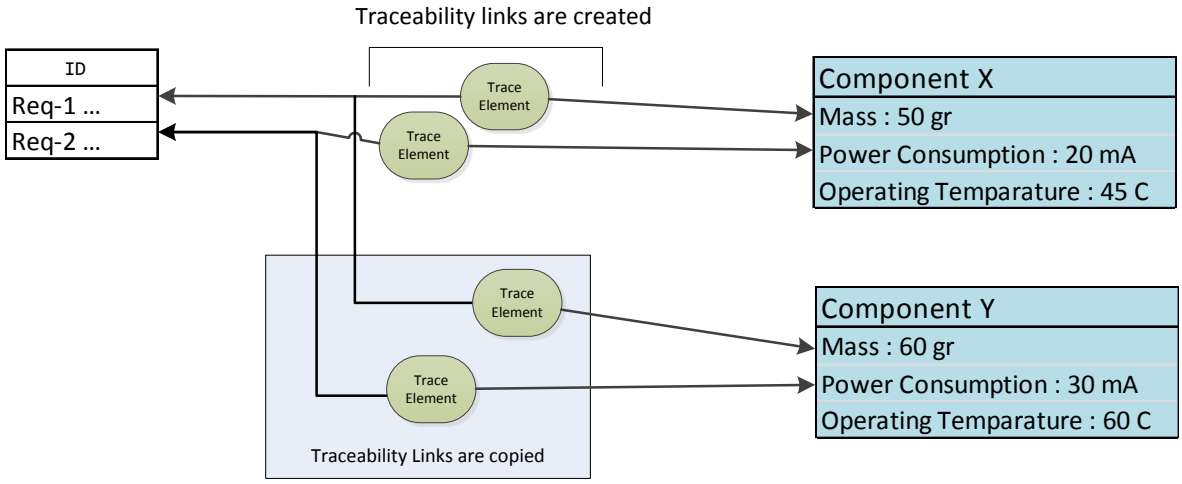


Figure 3.24: The traceability information can be automatically copied

artifact and the other model artifacts. However, this violation can also be detected after adding the existing artifact into the new model.

3.8.1 Obtaining Traceability Information for Artifact Reuse

In order to automatically determine the reusability of the existing artifacts, the traceability information between the new requirements and the existing artifacts should be present. However, when the requirements document for the new mission is obtained, there are not any traceability links between the requirements document and the existing artifacts. The traceability links can be manually created between the new requirements and for each of the existing components. However, with a large number of requirements and large number of components, it would be a time consuming task. This repetitive process can be automated, if the existing components are from the same type, and have same attributes. In such a case, manually creating the traceability links for a single component would be enough. For the rest of the components, creation of the traceability links can be automatized. An example of the procedure can be seen in Figure 3.24. From the manually generated traceability links, it can be deduced that, mass of the component is related with the first requirement. Similarly, power consumption of the component is related with the second requirement. With the procedure, traceability information can be automatically copied to all of the existing artifacts regardless of the artifact count. Time spent on creating the traceability links can be greatly reduced with this automation.

Challenges of Sub-System Requirements

If the existing requirements are about a single equipment in the spacecraft, the usage of automatic validation to find candidate equipment from the previous missions is relatively simple, since a single equipment is responsible from all of the requirements. However, if the requirements are about sub-systems, there are several challenges; Firstly, in sub-systems, a functionality can be performed by the combination of the several equipment. As an example, a requirement about supplied power

can be traced to multiple batteries and can be validated by taking the sum off the power generated by all the batteries. On the contrary, in another design, the same requirement could be satisfied by a single battery. In such a situation, the automatic generation of the traceability links would not work, since it is not possible to determine the amount of the batteries of the different designs. In such cases, the requirements should be manually traced with respected engines to the existing components.

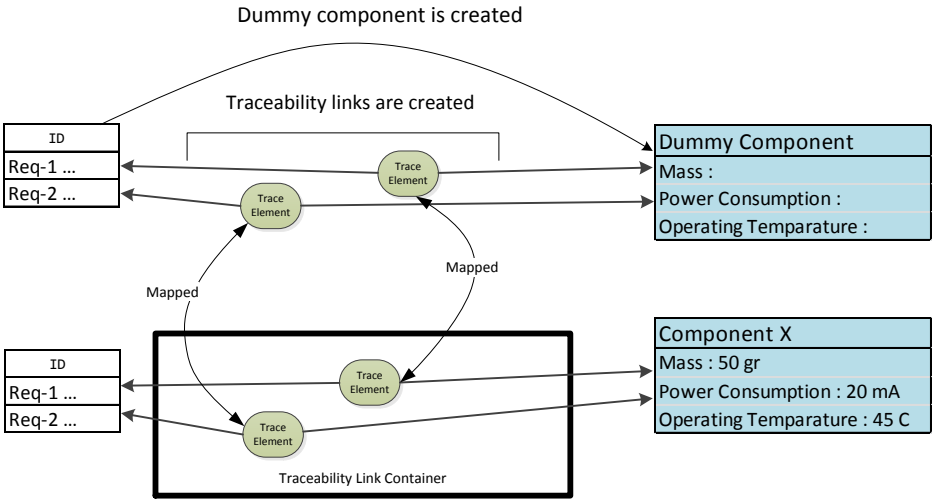
Secondly, a requirement can be violated due to an extra functionality in the sub-system. However, if that extra functionality is removed, than the design can be suitable for the new requirement. Consider the following example;

- The mass of the AOCS should be at most 3000 Grams.

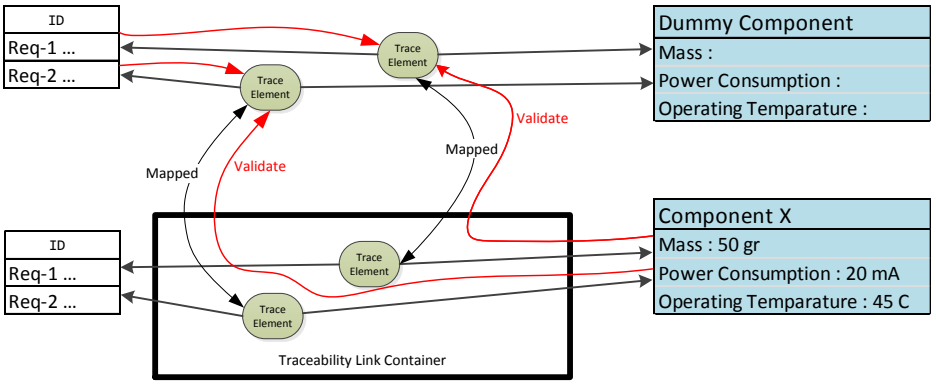
Consider an AOCS which satisfies all the other requirements except the given requirement. If AOCS contains an equipment which is not needed for the new mission, existence of this equipment increases the mass of the AOCS. Because of that, when compared, the mass of the AOCS would violate the requirement. However, this is not an actual violation. In order to avoid such situations, the existing design should be modified first, such as removing the unnecessary equipment. Then the reusability of the design can be assessed.

3.8.2 Comparing the Traceability Link Containers

Creating traceability links between the new mission requirements and the existing design artifacts may not be a desired situation. A traceability link would imply that the requirement is satisfied by the traced artifact. However, in this case, that would not be true, since the traceability links are created to find a suitable artifact. This situation can be avoided, if the requirements and the traceability information of the existing artifacts are available. The traceability link containers can also be automatically compared as long as the existing components are from the same type. However, when the new mission requirements are received, there is not a traceability link container to compare it with the traceability link containers of the existing artifacts. In order to create the traceability link container for the new requirements, our artifact generation technique can be used. First, a dummy component on the basis of the requirements can be generated. Since the traceability link would be also automatically generated, the relevant requirement for the artifact would be easy to detect. Then, without modeling the information, the requirements can be traced to respected attributes with the respected validation engines. Since the dummy component as well as the existing components are from the same type, the trace elements of different traceability link containers can be automatically mapped. Figure 3.25a depicts an example of how this mapping functions. In the example, the two trace elements which are related with mass are mapped and the trace elements which are related with power consumption are mapped with each other. An assumption here is that, in each requirements specification document, there should be exactly one requirement related with an attribute of the component, and no requirements should be conflicting with each other. When there are multiple trace elements related with the same attribute, this automatic mapping would not work. As depicted in Figure 3.25b, the attribute values of the existing component can be retrieved for the new requirements after the mapping. With this way, it is possible to determine how many requirements are satisfied by the each of the existing models, without creating direct traceability links between the new requirements and the existing model artifacts.



(a) Mapping between the trace elements



(b) The attribute values can be obtained through mapping

Figure 3.25: Comparing the traceability link containers

4 Implementation

In this section, we explain our prototype implementation regarding our methodology. We have implemented an Eclipse based tool called "ReqTrace" to store the traceability information and perform the automatic validation. The general structure of our implementation is depicted in Figure 4.1. ReqTrace is a UI oriented tool, which provides multiple wizards to perform the functionalities we have elaborated on the methodology. Requirements can be generated and modified with ReqTrace. In order to generate traceability links, ReqTrace has a *Traceability Wizard*, with the wizard, a traceability link can be generated between a requirement and a model element. From the requirements which are traced to model artifacts with automatic validation, it is possible to generate structured requirements. For the artifact generation, ReqTrace has the *Artifact Generation Wizard*. The wizard allows creation of model artifacts on the basis of a requirement, or a trace element. For the requirement-based artifact reusability, ReqTrace offers comparison functionality between model artifacts, as well as comparison functionality between traceability links belonging to different requirement documents. However, ReqTrace is not a standalone tool, it performs the mentioned functionalities when it is integrated with another application. The integration is done mainly through Eclipse's extension points. In Section 4.2 we explain how to integrate our tool with other eclipse based applications. In Section 4.3 we describe our design and implementation for providing automatic validation. In Section 4.4 we explain the architecture of our artifact generation implementation. In Section 4.5 we give details on the architecture of the requirement-based artifact reusability feature.

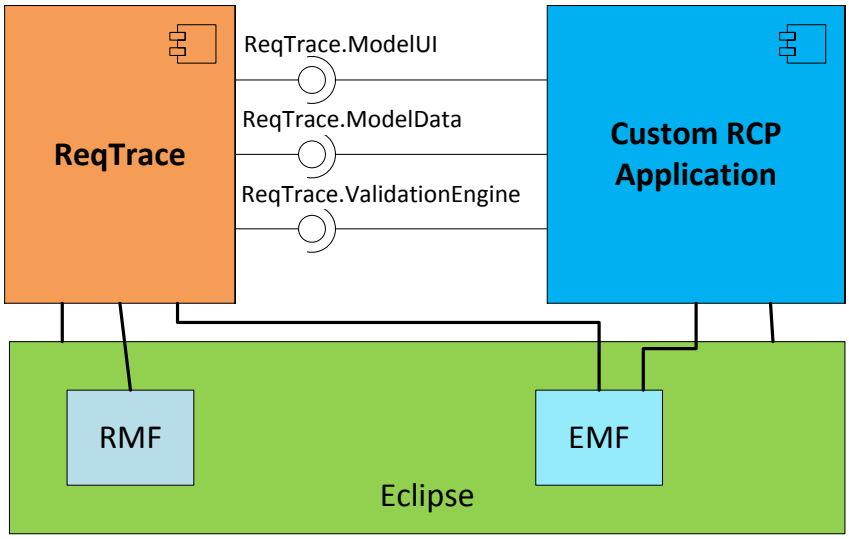


Figure 4.1: The general architecture of the implementation

4.1 Technologies used for the implementation

In order to provide traceability between requirements and model artifacts, we first need a mechanism to present the requirements. For that purpose we are using Requirements Management Framework (RMF). RMF is an open source framework reference implementation of the ReqIF standard. RMF contains primarily two components "RMF CORE" and "ProR". RMF CORE is the model implementation of the ReqIF standard using EMF. "Pror" is the GUI for displaying, modifying and generating new requirements. In our tool, we use the data model of RMF to create and store the requirements. The requirements are displayed with the editors provided by ProR. The structure of a requirement is flexible in the ReqIF standard. Each individual requirement is a *specObject*, and these *specObjects* can have multiple attributes with different types regarding the needs of the users. An example structure of a *specObject* can be seen in 4.1.1. Moreover, with the UI features from ProR, it is possible to structure the requirements, such as defining different specification types, adding attributes for individual requirements and defining enumeration elements for attributes such as "Status". RMF internally offers the capability for requirement to requirement traceability. On top of that, our tool contributes traceability between requirement to model artifacts. Since RMF is an open source software framework, developers around the world are currently contributing to it. However, since it is not a commercial tool, it contains some bugs which are relevant with our work. One bug we have seen so far is that, the editor for displaying the requirements is crashing unexpectedly. However, since this issue is not frequent and can be solved by simply closing and reopening the editor, we did not investigate the issue further.

Example 4.1.1 (Example *SpecObject* Attributes)**.**

- ID (String Type)
- Description (String Type)
- Owner (String Type)
- Creation Date (Date Type)
- Status (Enumeration Type with accepted and rejected)

4.2 Extension points and Extensions

Eclipse provides the functionality of extension points and extensions to avoid high coupling between plugins. A plug-in can declare an extension point to provide a functionality, but this functionality should be provided by the extensions. This ensures the loose coupling between classes and inversion of control. With the same idea in mind, our tool ReqTrace also defines some extension points to be filled by the application developers through extensions. ReqTrace defines three extension points, the structure and the content of the extension points is as follows;

- **ReqTrace.ModelUI** (The extension point for UI): ReqTrace provides different wizards for creation of traceability links, artifact generation and artifact reuse. Those wizards display the requirements and the model artifacts. However, ReqTrace has no information about how to

```
1
2  public interface IModelUIContentProvider {
3
4    // Methods to be used for creating traceability links
5    IContentProvider getContentProvider();
6    ILabelProvider getLabelProvider();
7    Object getInput(IProject project);
8
9    // Methods to be used for artifact generation
10   Object getArtifactGenerationInput(IProject project);
11   IContentProvider getArtifactGenerationContentProvider();
12   ILabelProvider getArtifactGenerationLabelProvider();
13   Object generate(Object owner, Object element);
14
15   // Ordering of a requirement Type
16   List<String> provideRequirementAttributeOrder(String reqType);
17 }
```

Listing 4.1.: The interface for registering UI content providers

display an arbitrary data model artifact. For that purpose, in order to use ReqTrace, application developers should implement and register an extension for *ReqTrace.ModelUI*. The extension should be a java class with the Interface *IModelUIContentProvider* implemented, which can be seen in Listing 4.1. The class should provide the content and the label providers to display the model elements. Moreover, in other to keep our tool general, extendable and easy to use, we do not limit the structure of the requirements. Therefore, each application using ReqTrace can define their own requirement structures with different attributes regarding the ReqIF standard. Unfortunately, there is not an ordering mechanism for the attributes of a requirement within the ReqIF standard. However, in order to display the requirements in the previously mentioned wizards, the order of the attributes are needed. Hence, the ordering of the attributes should also be provided in the extension by the application. Consider the example attributes in 4.1.1. If such a requirement needs to be displayed, ReqTrace can obtain the desired order through the registered extension.

- **ReqTrace.ModelData** (Extension point for getting data from the data model artifact). ReqTrace obtains the information about model artifacts through this extension point. In order to provide the data about the application model, an extension should be implemented and registered. The extension is a java class which implements the Interface *IModelElementDataProvider*. The interface can be seen in Listing 4.2. When displaying errors, artifact name is needed, or for the automatic validation, attributes of the artifact are needed. Moreover, data manipulations such as unit conversion should also be handled on the application side.

- **ReqTrace.ValidationEngine** (Extension point for registering new validation engines)

  Although ReqTrace offers some built in validation engines, each application may require different validation engines. Each custom validation engine should be implemented as an extension to this extension point. The helper classes implemented in ReqTrace can also be used by

```
1  public interface IModelElementDataProvider {
2
3    float getValue(EObject eObject);
4    String getUnit(EObject eObject);
5    String getObjectName(EObject eObject);
6    float getObjectValueAccordingToRequirementUnit(EObject eObject, String unit);
7  }
```

Listing 4.2.: The interface for obtaining model element data

```
1  public interface IValidationEngine {
2
3    boolean validate(TraceElement traceElement);
4    boolean canValidate(TraceElement traceElement);
5    String getValidationEngineName();
6    boolean canProvideSemantic();
7    String getSemantic();
8    String generateBoilerPlate(TraceElement traceElement);
9
10 }
```

Listing 4.3.: The interface for registering new validation engines

the application developers for the custom validation engines. The custom validation engines should implement the interface *IValidationEngine*. The interface can be seen in Listing 4.3.

The extensions for the first two extension points must be registered by the application developers, since without them it would not be possible to use the functionality of ReqTrace. Custom validation engines are optional to register, since in some applications the build in validation engines of ReqTrace can be enough to satisfy the users needs.

4.3 Validation

The general workflow of our automatic validation feature can be seen in Figure 4.2. To provide this feature we have implemented a traceability model, which we elaborate in detail in Subsection 4.3.1. Validation engines are responsible for performing the actual validation which we describe in Subsection 4.3.2. Moreover, to perform the automatic validation, we need a mechanism to detect the requirement changes as well as the changes on the model artifacts. We give details on the mechanism and our implementation in Subsection 4.3.4.

4.3.1 Traceability Model

To create the traceability between requirements and model artifacts, we have defined an Ecore model called *Trace Model*. The architecture of the *Trace Model* can be seen in Figure 4.3. In our implementation, each ".reqIf" file has a corresponding *TraceabilityLinkContainer*. The *TraceabilityLinkContainer* contains the *TraceElements*. These elements are responsible for maintaining the traceability information as well as performing the automatic validation with the validation engines. The elements have a reference to the requirement, references to target elements which are model artifacts.

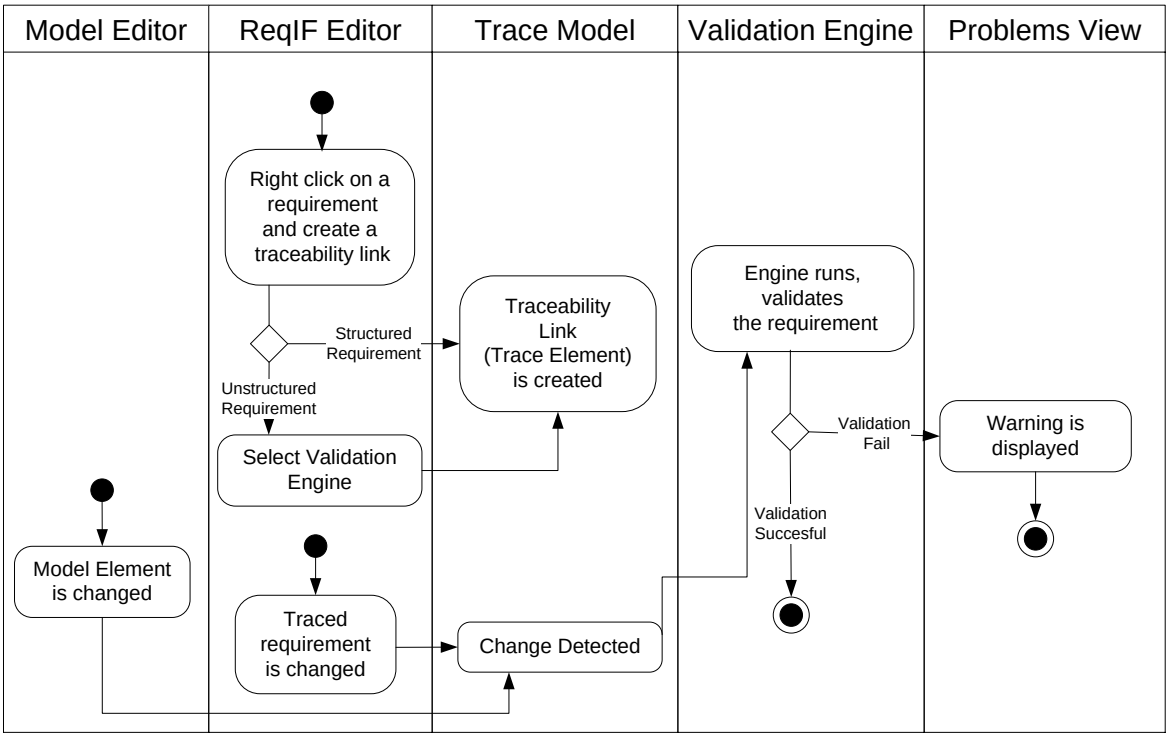


Figure 4.2: Activity Diagram of the Implementation

The trace elements have a name and description to describe their purpose. They also have a unique identifier. They store the name of the validation engine as an attribute. Intuitively, it makes sense to make this attribute an enumeration, however, enumeration definitions in Ecore cannot be extended. Since it should be possible to define different validation engines regarding the projects needs, this attribute implemented as String.

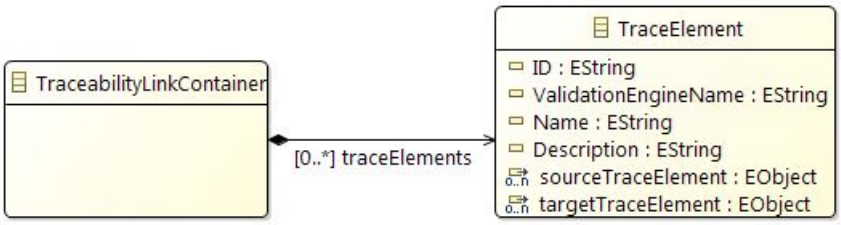


Figure 4.3: Traceability Model

4.3.2 Validation Engines Implementation

We have described how to register new validation engines in Section 4.2. Since numerical validation is one of the most common validation type, we have implemented some standard numerical validation engines in the ReqTrace. These are EqualsValidator, LessThanValidator and GreaterThanValidator. These validators only compare quantifiable information regarding the unit. As we have discussed in the methodology, unit conversions should be done by the data model which uses our

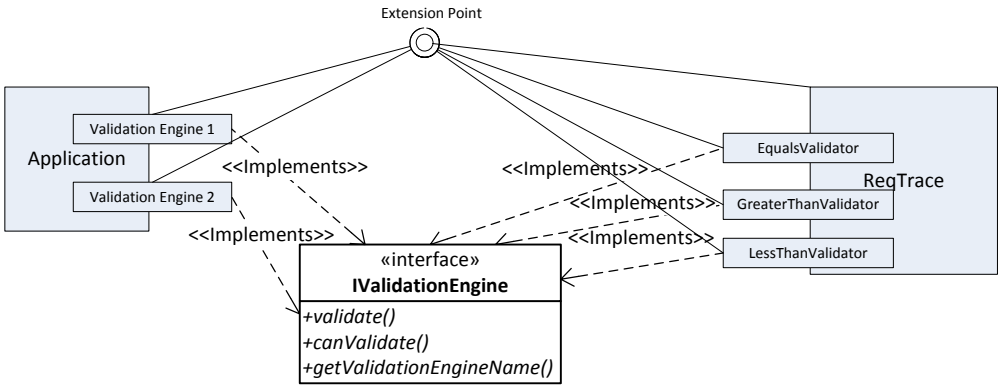


Figure 4.4: Engines Registered from Application and ReqTrace

tool. Moreover, we have implemented an abstract class for numerical comparisons which is called *NumericalValidationEngine*. This abstract class provides several functionalities. First one is to check unit compliance. As depicted in Figure 3.5, before the validation, the compliance of the unit given in the requirement and the unit used in the data model needs to be checked. This operation should not be individually implemented for every engine. Any new numerical validation engine can extend this class. Second functionality is that, as explained in Figure 3.12, the engine provides a boilerplate for the generation of structured requirements. If a new engine extends this class, no further programming is needed to generate structured requirements. Similarly, for validating the system wide numerical requirements, we have defined, *SystemWideNumericalValidationEngine*. Engines extending this class should provide the attribute they work on as well as the component type. Unit compliance and generation of structured requirements are handled within the class. As for the other type of validation engines, we have implemented the following: *EnumValidationEngine*, *ObjectDependentNumericalValidationEngine*, *ObjectDependentEnumValidationEngine*, *SystemWideEnumValidationEngine*. However, if the desired functionality cannot be achieved by extending those engines, the developers can implement their engines by extending *IValidationEngine*. New templates can also be provided with the custom engines. Figure 4.4 depicts the general structure. Then these custom validation engines can be registered to our tool using an extension point. When creating a traceability link, the tool obtains all the registered validation engines and provides them to the user. Then the user can select the desired validation engine.

4.3.3 Generating the Structured Requirements

In order to provide the generation of the structured requirements, we have provided a ".reqif" file template which has a specification called *boilerplate*. The specification should contain the *boilerplate type* requirements. These type of requirements has two attributes, namely "ID" and "Description". For the *ID* part, the same requirement *ID* is used to avoid confusion. For the simple requirements, ReqTrace generates the description part by combining the artifact name, artifact attribute value, engine semantic and the requirement attributes. For the system wide and object dependent requirements, attribute and component type is also received from the validation engine. According to the methodology, each structured requirement is generated regarding the templates given in

Subsection 3.6.1.

4.3.4 Detecting the Changes

An important step to implement our methodology is to provide a functionality to detect the changes on the requirements as well as model artifacts. In Eclipse, this can be done in two ways. The first one is to add listeners to resources. If a resource changes, resource listeners in Eclipse can detect it. We have tried to detect the requirement changes by implementing resource listeners, however we have encountered some problems. Firstly, RMF framework handles changes in a way that, it first modifies the *.reqif* file and unloads the resource. Then the framework re-loads the changed file to a new resource. In this process the listeners are lost. Secondly, since ReqTrace would be used with another project containing a data model, the artifacts of the project may use different resources which cannot be accessed. Then the tool could not obtain the changes on the model artifacts. Thirdly, providing persistency is also problematic with resource listeners, since the only way to provide the persistency is saving the workspace state. But ideally, the implementation should not depend on a workspace. Another mechanism to handle the resource changes in Eclipse is *Builders*. Regarding the problems of resource listeners, we have implemented a builder called *TraceBuilder*. To be able to add the builder to a project we have also implemented a Project Nature called *TraceNature*. Whenever some change in the project occurs, the builder obtains the changed requirements or model artifacts. If the changed requirement or the changed model artifact has a traceability link, the engine related of the traceability link runs automatically and an error is displayed if the validation fails. This approach works reliably as long as the model artifacts and the requirements document as well as the traceability link container are residing in the same project, even though they may use different resource sets. One shortcoming regarding this implementation is that, with the builder, we can obtain the changed resource, not the individual requirement. Consider a requirements document with a large number of requirements which are traced to model artifacts. If one of the requirements is changed, the builder would return the resource for the entire requirements document. With the validation engines, this is not a big issue, since automatic validation can be triggered for all of the requirements in that resource, and an error can be displayed for the ones which are not validated. This approach is not optimal in terms of performance because a single requirement change would trigger automatic validation on all requirements on that resource. However, this is not a big performance issue, since typically there should be a requirements document corresponding to a component. We do not expect thousands of requirements in a single requirements document. Moreover, this approach is necessary to detect the changes on the model elements. Otherwise, when a model element changes, a comprehensive search is needed to determine if the element is traced to requirement or not. By validating all requirements on a change, it is guaranteed that the changes on the model elements are also covered. However, unlike automatic validation, for the requirements which are traced with the *Inspection* engine, the individual requirement must be detected in order to give a proper warning which states *the individual requirement which is changed* and *the model artifact which needs to be reviewed.* In order to provide that capability, we have used EMF model compare. ReqTrace obtains the old resource from the local history, and the newly changed resource. By comparing them with *EMFCompare* the individual requirement which has changed can be detected. ReqTrace then displays a warning about the changed requirement, and the artifact which needs to be reviewed.

4.4 Artifact Generation

ReqTrace also offers the artifact generation functionality. Users can create an artifact on the basis of an existing requirement. Figure 4.5 depicts the sequence of the artifact generation. To allow users to select the artifact type and the location of the newly generated artifact we have implemented a Wizard called "Artifact Generation Wizard". The input for the artifact type selection should be registered by the application which uses our tool ReqTrace. ReqTrace then can obtain the necessary content providers and label providers to show the artifacts. The actual generation of the artifact should be handled on the application side, since adding new elements to the model may require different permissions and these permissions cannot be verified by ReqTrace. Moreover, it may not be possible to generate the desired artifact in the desired location. As an example, if the user tries to create a "Star Tracker" under the "Electrical Power System", the system should not allow it, since a star tracker is not related with the power system. This control mechanism can only be implemented in the application side. If the desired artifact can be created in the desired location, ReqTrace automatically generates a traceability link between the initial requirement and the newly generated artifact. The validation engine of the traceability link is set to *Inspection* per default. The name, description and the validation engine of the traceability link can be further modified by the user. If the artifact is generated on the basis of a *TraceElement*, the generated artifact is added to the targets of the *TraceElement*.

4.5 Requirements-Based Artifact Reuse

In order to perform requirements-based artifact reuse, we have implemented "ReusabilityWizard" in ReqTrace. The wizard displays the existing requirements and the model artifacts which have a traceability link from the requirements. Using the wizard, the user can select the desired requirements, and the artifacts which satisfy the selected requirements are automatically displayed. Similarly with the traceability wizard, the content provider and label provider for the wizard should be registered through an extension point. The order of the attributes of the requirements is also received through the extension point.

For finding the suitable artifacts, first the user needs to trace the requirements to one of the existing artifacts. In order to copy the traceability information to other artifacts, we have implemented the *CopyTraceabilityLinksWizard*. With the wizard, the desired artifacts can be selected and traceability links can be copied. For detecting the reusable artifacts, first, the tool obtains any artifact which is traced to a requirement in the given ".reqif" file, with the algorithm presented in Algorithm 4.1. These artifacts are then stored as candidate artifacts.

In the implementation architecture, the validation information between requirements and model artifacts are not stored. As explained in Subsection 4.3.4, validation engines are executed when a change in the system occurs. However, to determine the reusability of an artifact, the information about which artifact satisfies which requirement is needed. When this information is given such as in Table 4.1, then the intersection of the selected requirements can be simply taken.

In order to compute that information, the following two algorithms can be applied;

Algorithm 1: After the user selects a requirement, the artifacts which satisfy the requirement are computed from the candidate artifacts.

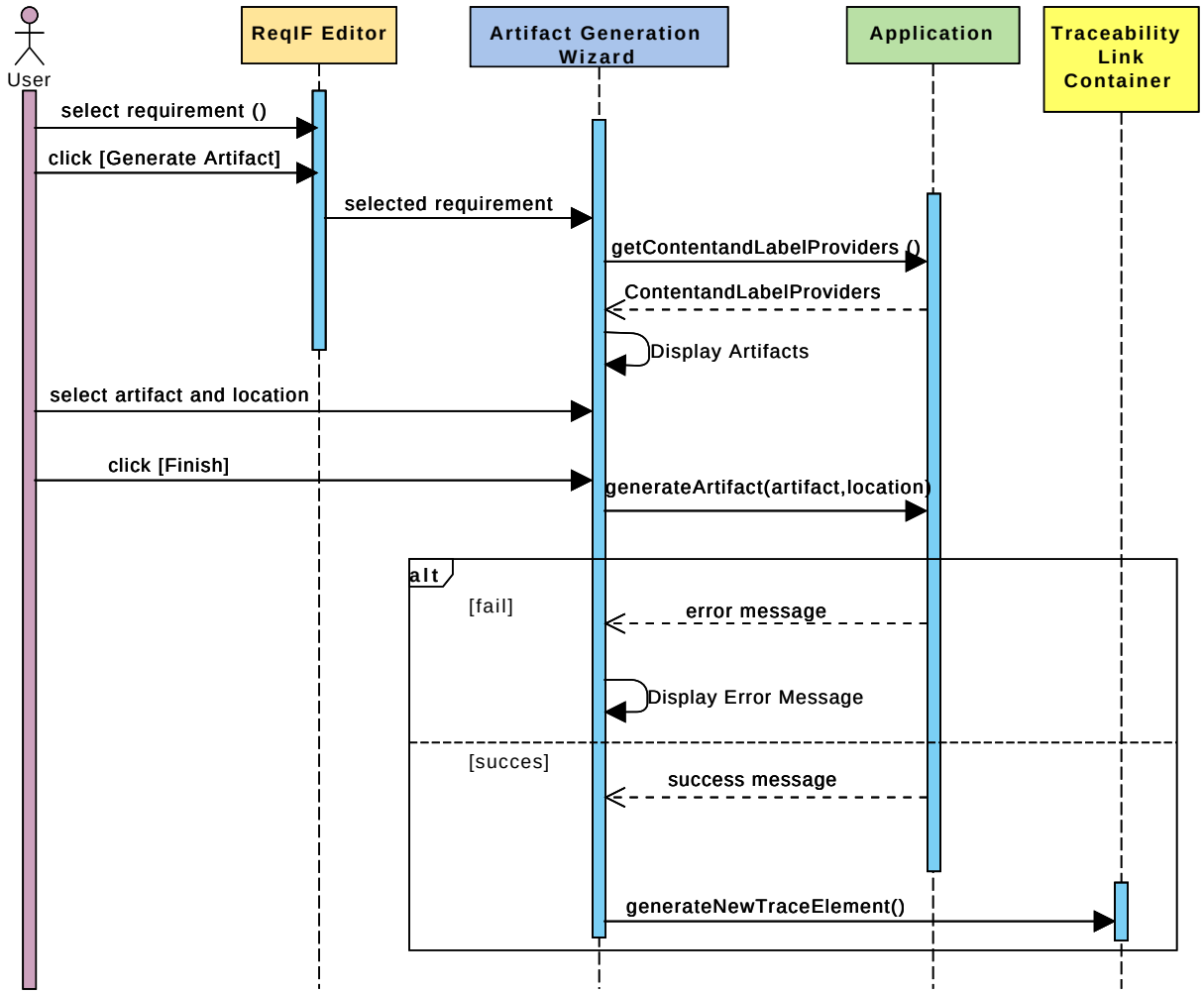


Figure 4.5: Sequence diagram of artifact generation

Input: *reqirementList*
Output: *candidateArtifactList*, which shows the possible candidates artifacts for satisfying the selected requirements from *reqirementlist*

```
1 method getCandidates(requirementList)
2   foreach requirementinreqirementList do
3     if requirementistraced then
4       candidateArtifactList.add(requirement.getTracedArtifact);
5     end
6   end
7 return candidateArtifactList;
```

Algorithm 4.1.: Finding the candidate artifacts

	Model Artifacts
Req-1	A, B, C
Req-2	B, C, D
Req-3	C, F, G

Table 4.1: Artifacts validating the requirements

	Traceability Link Containers
Req-1	TLC1, TLC2, TLC3
Req-2	TLC2, TLC3, TLC4
Req-3	TLC3, TLC5, TLC7

Table 4.2: Traceability Link Containers validating the requirements

Algorihtm 2: After the user opens the wizard and before any selection, for each requirement, the model artifacts satisfying the requirement are computed. This approach is applicable since the moment the wizard has opened, there is no possibility for the requirements or the model artifacts to change. Therefore, the validation information is always the same. Since the validation information for each of the requirements is already computed, intersection of the artifacts can be taken depending on the selected requirements.

The two approaches have performance advantages on each other depending on the usage scenario. Consider a requirements document with a large set of requirements, where all of the requirements are traced to at least a model artifact.

- **Scenario A**: The user only selects the first two requirements.
- **Scenario B**: The user selects various different requirement combinations multiple times.

The first algorithm performs better in Scenario A, since only the validation information for the first two requirements are computed. But with the second algorithm, the validation information for all of the requirements are computed, which is mostly redundant. On the contrary, in Scenario B, with the first algorithm, the same validation information for a requirement is computed multiple times. With the second algorithm, this recomputation is avoided since validation is performed only once per requirement before the wizard is displayed. Therefore, in Scenario B, second algorithm performs better. In order to combine the advantages of both of the algorithm, we have decided to use *HashMap* functionality of Java. After a user selects a requirement, the artifacts satisfying that requirement are computed and stored in a *HashMap*. Then if a deselection and a reselection on that requirement occurs, artifacts can be obtained from the *HashMap* without recomputing the validation information.

CompareTraceabilityLinkContainerWizard is implemented for comparing the *TraceabilityLinkContainers*. First, a traceability link container for the new requirements should be generated by the user, as explained in Section 3.8. Then the user can select the desired *TraceabilityLinkContainers* from a selection page for comparison. Similarly with comparing the artifacts, if a table as in Table 4.2 is available, depending on the requirement selection the intersection can be taken. This information can be obtained by mapping the trace elements of the existing design and the newly created trace elements. For the mapping, the attributes of the model element are used. As an example, trace elements related with the mass of a component are mapped together. Then by running the validation engine of the existing trace element with the new requirement attributes and data from the existing

design, it can be determined that if the existing satisfies the new requirement or not. However, similar performance concerns exist in this case as well. Thus, we apply the same algorithm explained before for the mapping and validation. When a requirement is selected, the *TraceabilityLinkContainers* which satisfy the requirement are added to the *HashMap*.

5 Evaluation

In Chapter 4, we have described the details about our tool implementation to provide the functionalities of our methodology. In this chapter, we provide an evaluation in order to assess the potential of our methodology. First, we provide a qualitative analysis by presenting how our tool can be used with a modeling software. The modeling software to be used in the evaluation is Virtual Satellite from DLR. In Section 5.1 we explain the integration between our tool ReqTrace and Virtual Satellite. In Section 5.2, we provide a quantitative analysis of our traceability methodology and automatic validation. Moreover, in Section 5.3 we evaluate our artifact generation methodology. Section 5.4 covers the evaluation of our methodology regarding requirements-based artifact reuse.

5.1 Virtual Satellite Integration

We have shown in Figure 4.1, that each application using ReqTrace must provide extensions for *modelData* and *ModelUI* to present and obtain the object information. In order to integrate our tool to Virtual Satellite, we have implemented the required extensions for the Virtual Satellite as it can be seen in Figure 5.1. Moreover, we have also implemented some validation engines regarding the features offered by Virtual Satellite. In a spacecraft, components are connected through interfaces to with other components. From our research, we have observed that, many requirements are specifying which interfaces should be used by each component. In Virtual Satellite, this information is modeled through with interface ends, and interfaces as can be seen in Figure 5.2. Moreover, components in a spacecraft have different system modes. As an example, in order to save energy, components enter the passive mode when they are not being used. Virtual Satellite models this behavior of the components by state machines. States represent the different modes of a component, and transitions between the states determine on which event the component should switch the mode. The engines implemented in Virtual Satellite are as follows;

- **Interface [AllOf] Validator** The engine checks if the component can be connected by all the interfaces defined on the requirement.
- **Interface [OneOf] Validator** The engine checks if the component can be connected by one of the interfaces defined on the requirement.
- **Modes Validator** The engine checks if the component supports the modes defined on the requirement.
- **Active Mode Validator** The engine checks if the traced component of the spacecraft is active during a given mission mode.
- **Passive Mode Validator** The engine checks if the traced component of the spacecraft is passive during a given mission mode.

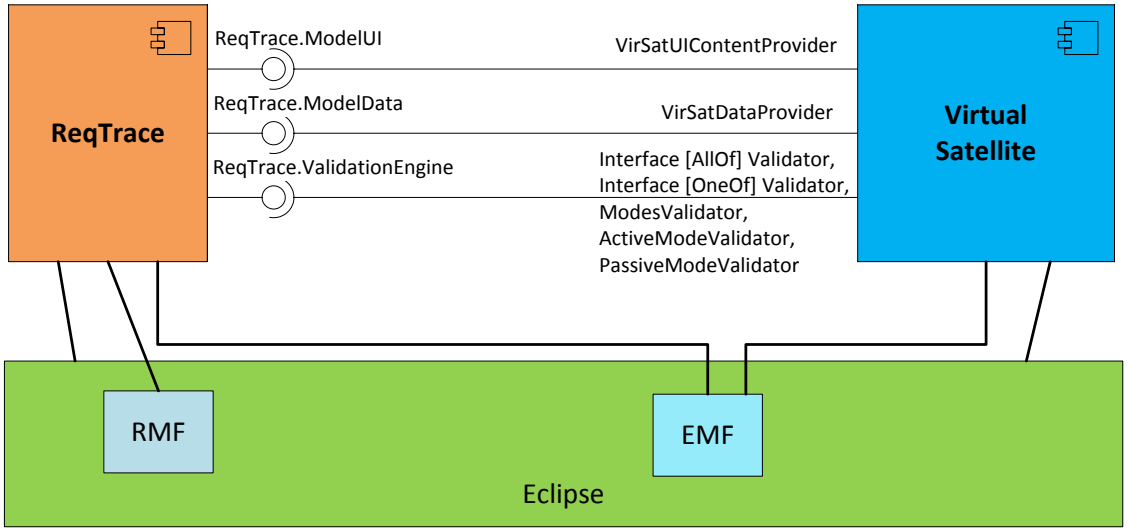


Figure 5.1: Integration of ReqTrace and Virtual Satellite

```
1 public class InterfaceAllOfValidator implements  IValidationEngine {
2 private static final String VALIDATIONENGINENAME = "Interface [AllOf] Validatior";
3   @Override
4   public boolean validate(TraceElement traceElement) {
5     String reqValue = TraceValidatorHelper.getValueWithColumnName(traceElement, "Enum");
6     // do the validation
7   }
8 }
```

Listing 5.1.: Example structure of the validation engines implemented and registered in Virtual Satellite

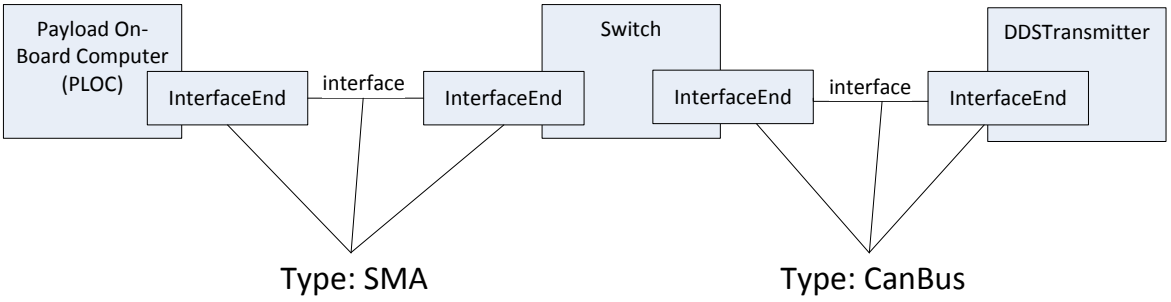
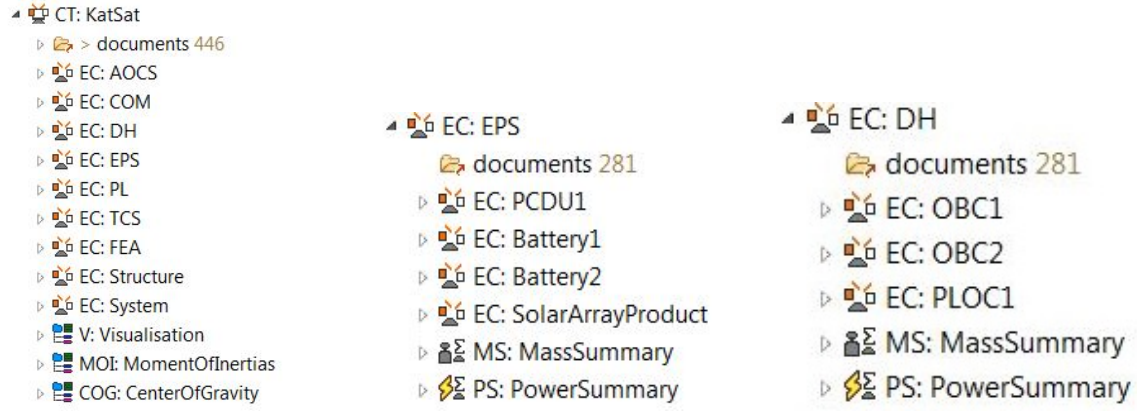


Figure 5.2: The components have interface ends and connected through interfaces



(a) Sub-systems of the spacecraft

(b) Equipment of electrical power system

(c) Equipment of the data handling system

Figure 5.3: Sub-systems and equipment of the spacecraft

5.1.1 Data Model for a Conceptual Spacecraft

We will use a data set which was developed in DLR to evaluate our methodology. The data set is a conceptual spacecraft model, which is called "KatSat". The main objective of the spacecraft is to detect forest fires from space via an infrared camera. The data set is developed by an aerospace engineer in DLR and used mainly to test the functionalities of the Virtual Satellite. The spacecraft model consists of seven sub-systems which can be seen in Figure 5.3a, a payload which carries the infrared camera and in total of forty eight equipment. The example equipment for the sub-systems can be seen in Figure 5.3b and Figure 5.3c. However, there are not any requirements for that conceptual spacecraft model.

5.1.2 Generating the Requirements

In order to perform the evaluation, we need the requirements for the spacecraft model given in the previous section. If the requirements for a similar mission can be obtained, the necessary modifications can be made to create the requirements for the data set. Unfortunately, finding real requirements for space missions is not an easy task, since most of the space projects are to some degree confidential. However, we were able to find three different information resources to create meaningful and representative requirements to evaluate our methodology. Firstly, we were able to contact with the engineers in DLR Bremen and from them, we were able to obtain mission requirements for S2TEP. Secondly, we take basis from the book Space Mission Engineering the new SMAD [39] which conceptually explains the space missions and it also contains some example requirements from older space missions such as "FireSat II". Thirdly, for the equipment requirements, we have analyzed the commercial spacecraft products. From the manifacturers product description flyers, we were able to deduce what the product offers and the possible requirements. As an example, we have gathered possible star tracker requirements by looking different product specifications such as [50]. From those resources we have generated 84 requirements for 10 different equipment, 8 requirements for the infrared camera, 2 requirements for the general mission, and 6 requirements for the harness(cables connecting the components). In total we use a set of 100 requirements. Although we expect much higher amount of requirements for a real mission, these requirements

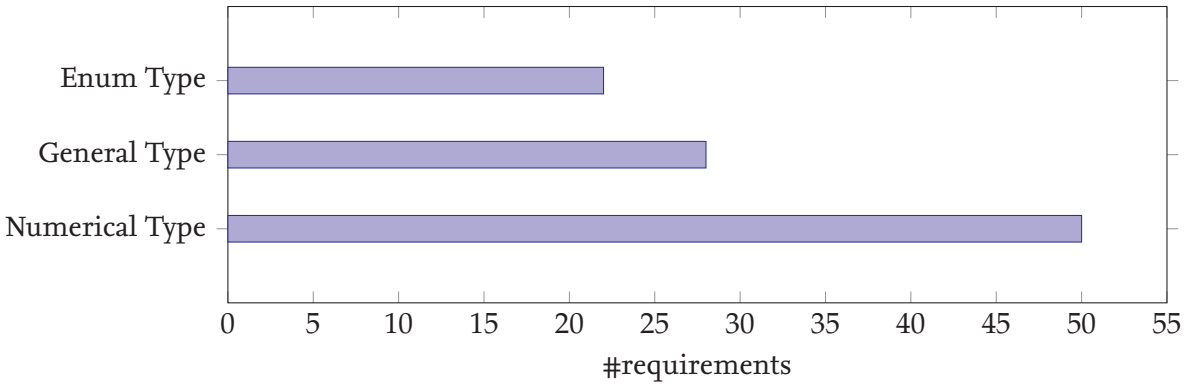


Figure 5.4: The amount of requirements for each requirement type

would be enough to give an overview about the strengths and weaknesses of the methodology. The requirements for the data set can be found in Section A.1.

Formatting the Requirements

As discussed in the Section 3.2, we present the *Value* information in the requirement attributes for the automatic validation. Depending on the *Value* information, we have divided the requirement set to three types, which are numerical type, enum type, and general type. The requirements which contain some numerical value and unit are reformatted, where the numerical value and the unit are presented in different requirement attributes. Requirements providing listed enumeration are presented under enum type, where the enumeration information is presented in a requirements attribute. The rest of the requirements are written under the general type. The distribution of the requirement types in our requirement set can be seen in Figure 5.4.

5.2 Evaluation of Automatic Validation

In order to show how our implementation can contribute to a modeling software, we perform a qualitative analysis first. The usage of our tool with Virtual Satellite is depicted in Figure 5.5. In the example, requirement 6 is traced with automatic validation. Since the model artifact is not satisfying the requirement, an error is displayed to the user. Furthermore, requirement 5 is traced with *Inspection* engine. Since the requirement has changed recently, a notification is provided to the user.

With our methodology, requirements can be automatically validated, and when a requirement changes, the impact of the change on the model artifacts can be automatically determined. In order to measure to what degree these functionalities can be provided, and to perform a quantitative analysis, we use the KatSat data set and the requirements we have provided. With the requirements

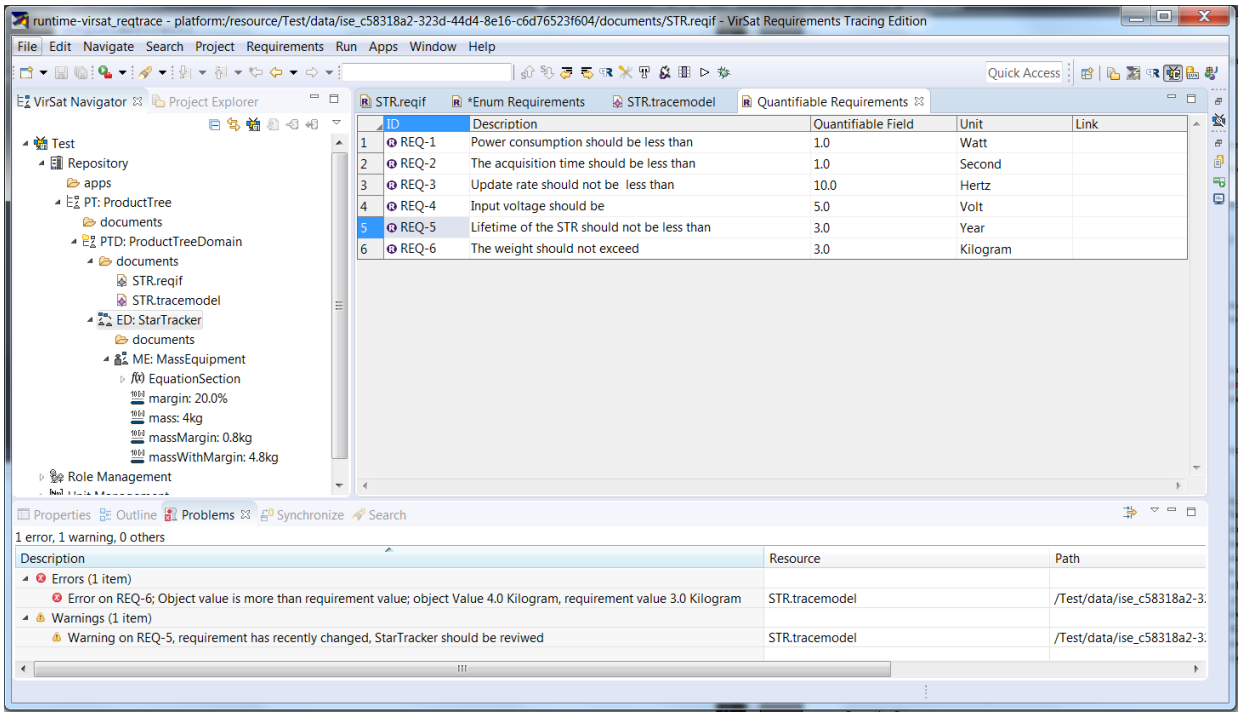


Figure 5.5: Requirement violations and the artifacts which should be reviewed are displayed

and the data set, we aim to obtain the following metrics;

- **Validation Rate of ReqTrace Engines**: Since the mission requirements contain large amount of numerical information, we have implemented some simple numerical comparison engines in ReqTrace for validating these requirements. What percentage of the requirements for the KatSat can be automatically validated by ReqTrace engines?

- **Validation Rate of Custom Engines**: ReqTrace also provides the capability of registering new engines, depending on the need. What percentage of the requirements for the KatSat can be automatically validated by the custom engines?

- **Validation Rate**: With the combination of ReqTrace engines and custom engines, what percentage of the requirements for the KatSat can be automatically validated?

- **Precision**: When some of the existing requirements are changed, what percentage of the violations are automatically detected?

- **Recall**: Among the detected violations, what percentage of them are true violations?

In Section 3.6, we have provided a methodology for generating structured requirements. For a quantitative analysis, we aim to find the generation rate of the structured requirements. Moreover, we have claimed that efficiency of the automatic validation can be increased by using structured requirements. Thus, we aim to compare the validation performance of the structured and unstruc-

tured requirements. For these purposes, the following metrics are needed;

- **Generation Rate:** From what percent of the initial requirements, structured requirements can be generated ?
- **Precision:** When the structured requirements change, what percentage of the violations are automatically detected?
- **Recall:** Among the detected violations, what percentage of them are true violations?

5.2.1 Metrics for Unstructured Requirements

The results for the unstructured requirements are as follows;

ReqTrace Engines

In total, 18% of the requirements are traced with ReqTrace engines. All of the requirements are from numerical type. As it can be seen in Figure 5.8, 36% of the numerical type requirements are traced with *GreaterThanValidator*, and the remaining 64% traced with the *Inspection* engine.

Engines Implemented in Virtual Satellite

In total, 26% of the requirements are traced with custom engines. All of the requirements are from enum type. The disribution of the engines in enum type requirements can be seen in Figure 5.8. In particular, approximately 80% of the enum type requirements are traced with an automatic validation engine, whereas the renaming 20% are traced with *Inspection* engine.

Validation Rate

In total, 44% of the requirements are automatically validated, whereas 56% of them are traced with *Inspection* engine while leaving the validation to the user as it can be seen in Figure 5.6.

Precision and Recall

Since the requirements are not from a real space mission, where requirements change naturally, we have asked an engineer from DLR, to make some sensible changes to the requirements. The limitations were; not to the change the requirement id, not to change the *Subject* of the requirement and not to change the requirement to something meaningless. The engineer has sufficient knowledge in requirements engineering and spacecraft design, however no knowledge about our thesis topic. We have excluded the requirements traced with *Inspection* engine, since regardless of the change, the outcome would be just a notification as depicted in Figure 5.1. Hence, he has changed the 44 requirements which are traced with automatic validation. The changed requirements can be seen in Section A.2. From the changed requirements, 29 of them were violated by the existing design, whereas 13 of them were still satisfied by the design. With our automatic validation, 19 of those violations are caught. Moreover, all of the detected violations were true violations. With these results, the precision is 100% since there are no false positives, and recall is 65% since only 19 of the violations caught over 29.

5.2.2 Metrics for Structured Requirements

The results for the structured requirements are as follows;

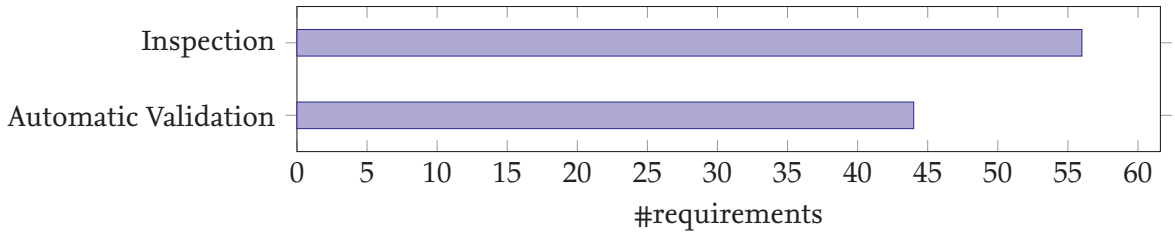


Figure 5.6: The distribution of automatically validated requirements

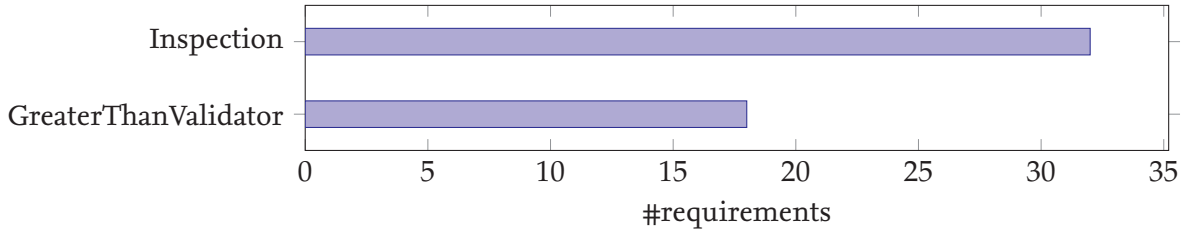


Figure 5.7: The distribution of engines on Numerical Type requirements

Generation Rate

Structured requirements are generated from 44% of the requirements. The generated requirements can be seen in Section A.3

Precision and Recall

Most of the changes cannot be directly applied to the structured requirements, since many of them contain negative sentences and many of them are not obeying the predefined syntax. Hence, we have applied the changes by keeping its semantic, but changing its syntax. (E.g, Instead of writing "Component X should not be active ....", we wrote "Component X should be passive..."). When we applied the changes to the structured requirements, we have observed that, all of the violations are automatically caught. Moreover, all of the detected violations are true violations. This would give 100% precision and 100% recall. The changed structured requirements can be seen in Section A.4.

5.2.3 Discussion

From the data we have obtained, the following deductions can be made.

Validation Rate

We have observed that, the automatic validation of the requirements strictly depends on the capabilities of the data model. *Inspection* engine is the most used engine, since the requirements which do not provide any information suitable for automatic validation have to be traced with *Inspection* engine. Moreover, if the information provided in the requirement is not modeled, these requirements are also traced with *Inspection* engine. It can be seen that, all of the numerical type requirements are suitable for automatic validation. However, the spacecraft model does not contain many of the presented information. We can deduce that, if the data model contains more information, more requirements from that type can be automatically validated. One interesting result is that, only 36% of the numerical type requirements are validated automatically, whereas 80% of the enum

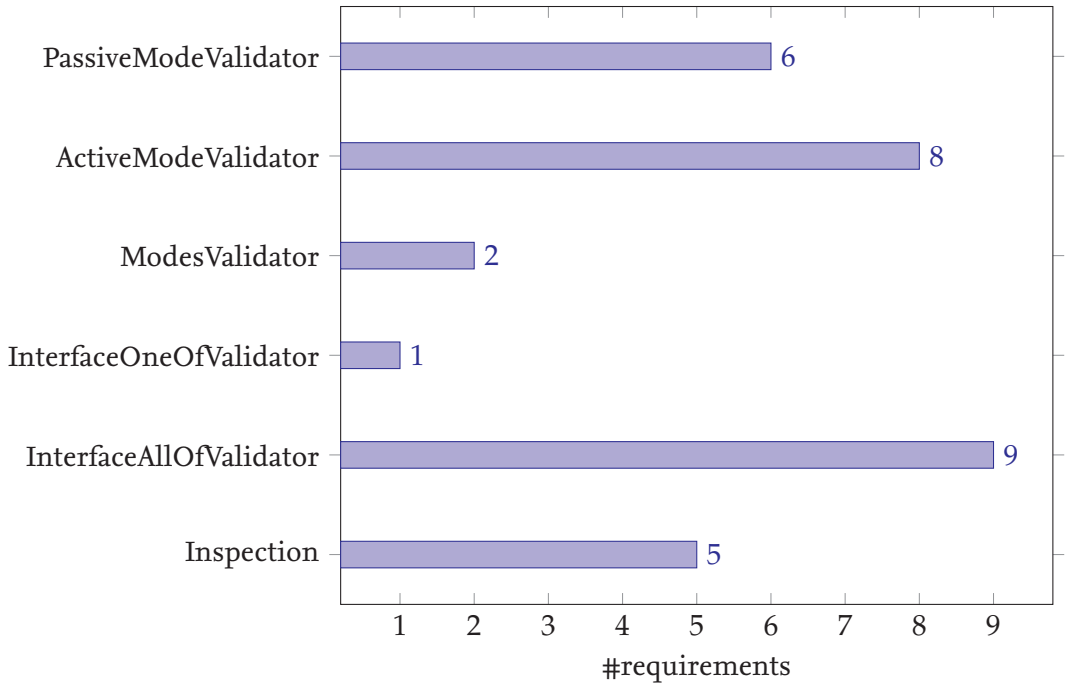


Figure 5.8: The distribution of engines on Enum Type requirements

type requirements are validated automatically. This can be strictly attributed to the data models functionality. The restrictions on modes and interfaces of a component is very common in requirements. Since these functionalities are well modeled in Virtual Satellite, validation rate of these requirements became higher. Moreover, for the ReqTrace engines we have seen that, *EqualsValidator* and *LessThanValidator* are not used for automatic validation. However, intuitively these engines are as useful as *GreaterThanValidator*. This situation has occurred because, in Virtual Satellite, mass and power consumption of the components are modeled. In general, a lighter component is more desirable than a heavier component since a lighter spacecraft would be easier to launch. Similarly, power consumption of a component is desired to be as low as possible, since power is scarce and valuable in the space. Because of that, requirements enforce a maximum limit on those attributes. However, for some other attributes, such as temperature tolerance, requirements pose a minimum value. We were not able to trace such requirements to model artifacts with automatic validation, because the temperature tolerance of a component is not modeled in Virtual Satellite. As for the custom engines, *InterfaceInterface[AllOf]Validator* is used more than *Interface[OneOf]Validator*. The reason we observe is that, most of the requirements are strictly forcing an interface type of a component. In rare occasions, a list of possibilities provided. Which decreases the usage of this engine. However, a deduction about the usefulness of an engine cannot be made with these data. Some engines may be used a few times, but as long as they contribute to the safety of the mission, they are useful. As can be seen in Figure 5.8, *ModesValidator* used only twice in our evaluation. However, it is actually validating the modes of the spacecraft. In this case, the main objective of the spacecraft is to provide fire detection functionality. Validation of the other requirements are meaningful, as long as the spacecraft provides fire detection functionality.

Precision and Recall

We have observed that, all the changes made on the *Value* part of the requirement are correctly caught with the validation engines. The missed ones are the semantical changes, in other words, the changes made on the *Action* part of the requirement, such as negating the requirement. This point is the most significant weakness of the validation of the unstructured requirements.

Generation Rate of the Structured Requirements:

The structured requirements could only be generated from the requirements which are traced with automatic validation. This generation rate also strictly depends on the capabilities of the data model. When the model has more information, more requirements can be traced with automatic validation, therefore, more structured requirements can be generated from the initial requirement set. In terms of readability, there is not a big difference between unstructured and structured requirements. The structured requirements are easy to understand and carry the same information with the original requirements. Hence, using the structured requirements would not cause any information loss in projects.

Precision and Recall of the Structured Requirements:

From the results we have obtained, it is possible to conclude that, the usage of structured requirements is not decreasing the automatic validation performance. Precision is same with the unstructured requirements, but we observe a huge increase in recall. This is due to the fact that, structured requirements can also detect the changes in *Action* part of the requirement. Moreover, structured requirements can be traced without selecting a validation engine, instead engine is automatically assigned. Tracing the requirements without selecting the engine is another important benefit, since during our evaluation we have observed that, there is a possibility that some of the requirements would be traced with an incorrect validation engine. This room for mistake is also eliminated with using the structured requirements.

In general, we observe that, as long as the data model has the information, the requirements can be automatically validated with our methodology. Automatic validation rate strictly depends on the requirement set and the functionality provided by the data model. The biggest weakness of the automatic validation is detecting the changes in *Action* part of the requirement. However, this weakness can be overcome by using structured requirements. Moreover, we have claimed that it would be more beneficial to generate and share the structured requirements instead of sharing the requirements directly. Both of the metrics support our claim. The structured requirements do not have a disadvantage over unstructured requirements and they can be more efficiently validated.

5.3 Evaluation of Artifact Generation

We evaluate artifact generation with the same requirements we have used in Section 5.2. We perform a qualitative analysis to determine which artifacts can be generated on the basis of the requirements. Where possible, we will use the same data set from Subsection 5.1.1. The scope of the evaluation is the use cases which we have defined in Section 3.7;

- Generating a new model on the basis of requirements
- Generating artifacts from the newly added features

- Generating artifacts on the requirement change
- Generating artifacts in a deeper level
- Generating artifacts on the basis of trace elements
- Generating budgets on the basis of requirements

Artifact Generation for a New Spacecraft Model

In order to evaluate this scenario, we have generated a new spacecraft model from the requirements in Subsection 5.1.2. The first requirement to start modeling is that **SAT-SYSTEM-001**, from the requirement, the root of the spacecraft model is created. From the remaining requirements we have generated the respected sub-systems. The generated artifacts on the basis of requirements can be seen in Table 5.3.

Requirement ID	Generated Artifacts
SAT-SYSTEM-001	Spacecraft
SAT-OBC-008	Data Handling
SAT-BAT-004	EPS
SAT-GPSREC-011	AOCS
SAT-HAR-006	Harness

Table 5.1: Artifacts generated from the requirement

Requirement ID	Related Artifact
SAT-BAT-001	Battery
SAT-SunSen-005	Sun Sensor
SAT-GPSANT-002	GPS Antenna
SAT-PCDU-002	PCDU
SAT-IRCAM-005	Infrared Camera
SAT-STR-004	Star Tracker

Table 5.2: Interface ends generated for the artifacts

Generating Artifacts from the Newly Added Features

As we have seen in Section 5.2, a significant portion of the numerical type requirements could not be automatically validated since the data model did not contain the necessary information. As we have further investigated, we have found that a new feature to model the voltage values was implemented in Virtual Satellite. The operating voltage of a component can be presented through power interface ends. Using the new feature, we were able to create power interface ends on the basis of those requirements. The requirements and related artifacts can be seen in Table 5.3.

Requirement ID	Generated Artifact
SAT-OBC-006	New Mode
SAT-DDS-006	New Interface End
SAT-TTC-007	New Interface End
SAT-SunSen-006	New Interface End
SAT-STR-009	New Interface End

Table 5.3: Artifacts generated after the requirements change

Requirement ID	Generated Artifact
SAT-SunSen-009	Sun Sensor
SAT-STR-003	Star Tracker
SAT-TTC-009	TTC Antenna
SAT-RW-003	Reaction Wheel
SAT-GPSANT-008	GPS Antenna
SAT-GPSREC-011	GPS Receiver

Table 5.4: The generated lower level artifacts from the respected requirements

OBC1		
Mass:	1,5	kg
Min Power:	5	watt
Max Power:	30	watt
Avg Power:	10	watt
Interfaces:	UART, SMA	

OBC2		
Mass:	1	kg
Min Power:	50	watt
Max Power:	100	watt
Avg Power:	60	watt
Interfaces:	UART, SMA, I2C, RS485	

OBC3		
Mass:	3,2	kg
Min Power:	6	watt
Max Power:	26	watt
Avg Power:	10	watt
Interfaces:	UART, SMA, I2C, RS485	

OBC4		
Mass:	1,9	kg
Min Power:	11	watt
Max Power:	24	watt
Avg Power:	16,2	watt
Interfaces:	I2C, RS485	

OBC5		
Mass:	1,8	kg
Min Power:	14	watt
Max Power:	34	watt
Avg Power:	18	watt
Interfaces:	UART, SMA	

OBC6		
Mass:	2,5	kg
Min Power:	7	watt
Max Power:	45	watt
Avg Power:	22,2	watt
Interfaces:	I2C, RS485	

OBC7		
Mass:	1,3	kg
Min Power:	6	watt
Max Power:	12	watt
Avg Power:	8,4	watt
Interfaces:	UART, SMA	

OBC8		
Mass:	3,5	kg
Min Power:	11	watt
Max Power:	32	watt
Avg Power:	19,4	watt
Interfaces:	I2C, RS485	

Figure 5.9: The on-board computer models from previous missions

Generating Artifacts in a Deeper Level

In the new spacecraft model, after the generation of sub-systems, we have generated the individual equipment of the sub-systems. Generated artifacts and corresponding requirements can be seen in Table 5.3.

Generating Artifacts on the Requirement Change

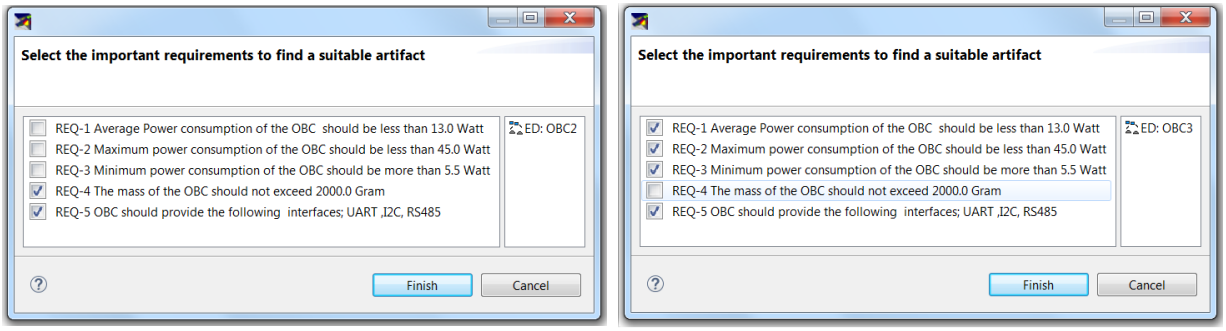
As it can be seen in Section 5.2, the changes on some of the requirements are not validated. The data set needs to be modified in order to be consistent with the requirements. We were able to update the data set by generating artifacts on the basis of the changed requirements. The generated artifacts and their requirements can be seen in Table 5.3.

5.3.1 Discussion

From the results we have observed that, artifact generation can be performed on each of the uses cases we have defined, if the proper requirements are in before hand. The spacecraft can be modeled by generating the sub-systems and equipment on the basis of requirements. Furthermore, artifact generation can be used to handle the changes on a requirement. We were not able to evaluate the remaining two use cases since we did not have the relevant requirements. For the budget generation, a proper requirement to generate the budgets of the components was not given in the requirement set. Similarly, for the artifact generation on the basis of trace elements, a relevant system wide requirement was not given.

5.4 Evaluation of Requirements-Based Artifact Reuse

In order to evaluate requirement-based artifact reuse, first, we perform a qualitative analysis by showing the functionality on a hypothetical scenario, where there are requirements for the on-board computer of the spacecraft which is required for the new mission and some existing on-board computer models. For the analysis, a small set of requirements for the on-board computer(obc) for the new mission is provided, which can be seen in Table 5.5. As for the existing models, 8 different



(a) OBC2 satisfies REQ-4 and REQ-5

(b) OBC3 satisfies maximum number of requirements

Figure 5.10: Requirement-Based Artifact Reuse

on-board computer models from the previous missions are provided. The attributes of the on-board computer models can be seen in Figure 5.9. We provide two scenarios. First, where the power consumption of the on-board computer is not a concern, an on-board computer model is needed which should satisfy the fourth and fifth requirement. This can be achieved efficiently as depicted in Figure 5.10a. In the second scenario, where all of the requirements are equally important, an on-board computer model which satisfies maximum number of requirements are required. This model can also be efficiently found, as it can be seen in Figure 5.10b.

Requirements-based artifact reuse provides two main benefits to the engineers. First, engineers can find the re-usable artifacts faster when compared to manual inspection. Secondly, manual inspection is prone to errors if there are large amounts of requirements and artifacts. For those reasons, we have provided two different approaches for finding the reusable artifacts automatically in the methodology. In order to measure the improvement on both, we perform a quantitative analysis. We have conducted an informal user study. The setup of the study and provided material can be seen in Table 5.6. The analysis is conducted with 9 participants, where participants are required to find suitable on-board computer models for the new mission with the provided scenarios. The participants are divided in to three groups. All groups are provided with the existing on-board computer models and the requirements for the desired on-board computer. First group is required to find the artifacts manually without the ReqTrace support. The second and third group are provided with the ReqTrace. Additional to the new requirements and existing models, third group is provided with the requirements for the existing on-board computer models, as well as the traceability link containers. For each of the participants, we have measured the time required to find the suitable models. Moreover, the incorrect answers are also noted in order to measure the accuracy. The results of the analysis can be seen in Table 5.7.

5.4.1 Discussion

Our informal study reveals that, requirement-based artifact reusability significantly reduced the time spent on finding a suitable artifact. The participants finding the models manually spent more time than participants using ReqTrace. Moreover, for the second group, a big portion of the time is spent on creating the traceability information between the requirements and the existing models. The third group also spent a similar time to create the traceability link container for the new

ID	Description	Value	Unit
Req-1	Average Power consumption of the OBC should be less than	13	Watt
Req-2	Maximum power consumption of the OBC should be less than	45	Watt
Req-3	Minimum power consumption of the OBC should be more than	5.5	Watt
Req-4	The mass of the OBC should not exceed	2000	Gram

ID	Description	Enum
Req-5	OBC should provide the following interfaces;	UART ,I2C, RS485

Table 5.5: The requirements for the on-board computer

	Provided Material	Methodology to find suitable models
Group 1	- Requirements for the OBC - 8 different OBC models from previous missions	Manual inspection
Group 2	- Requirements for the OBC - 8 different OBC models from previous missions - ReqTrace	Tracing the requirements to exising models
Group 3	- Requirements for the OBC - 8 different OBC models from previous missions - Requirements documents for the existing OBC models - Traceability Link Containers of the existing models - ReqTrace	Comparing the Traceability Link Containers

Table 5.6: The setup for the user study

	Group 1			Group 2			Group 3		
Participants	p1	p2	p3	p4	p5	p6	p7	p8	p9
Faults	2	0	1	0	0	0	0	0	0
Finished Tracing	n/a	n/a	n/a	2.29	2.41	2.51	2.18	2.25	2.49
Model Found	6.27	8.04	7.17	3.01	3.18	3.30	2.47	2.59	3.38

Table 5.7: The result of the user study

requirements. In a scenario, where the traceability information already exists, the difference would be much more significant. However, there is not a significant difference in terms of time reduction between the two approaches since the second and third group required similar amount of time to find the suitable artifacts.

For the accuracy of the answers, it can be observed that manual detection is prone to errors. In a real modeling environment, these wrong answers may lead to usage of a incompatible equipment. Moreover, besides using an incompatible equipment, a suitable model which can be reused may not be detected manually. This situation would increase the cost of the project, since it would force the engineers to model the equipment from the ground up, which would be a redundant work. On the contrary participants in the second and third group gave correct answers. This is an indication that, reusable artifacts can be found more reliably with the tool support.

5.5 Threats to Validity

In this section, we discuss about the potential threats for our evaluation. We categorize the threats into three groups. Construct validity, the threats regarding the design of the experiment. Internal validity, the threats regarding the causal relationship with the experiment data and the outcome, and external validity, the threats regarding the generalization of our results.

Construct Validity

We have evaluated our methodology with a conceptual spacecraft model by providing possible requirements for the model. For each of the cases, the results would be more reliable, with requirements for a real mission and a spacecraft model for the mission. For the artifact generation, we have analyzed the possible use cases with a conceptual spacecraft model. Although we have shown that the mentioned scenarios are applicable, a better evaluation should be conducted on a real modeling environment. In such an environment, we can determine that, on which of the scenarios artifact generation provided the most benefit by observing the engineers. Similar threat also exists for our evaluation for requirement-based artifact reuse. We need to obtain the measurements in a real modeling process, where there are real requirements and real model artifacts from previous missions. However, due to the confidentiality of the space missions, it will not be possible for us to obtain access for spacecraft models in ESA or NASA. However, our evaluation provides an overview about the potential of our methodology.

Internal Validity

For the interval validity, we identify the following threats.

- **Amount of the Requirements:** The precision and recall values are obtained through in total of 100 requirements for different equipment. However, depending on the mission, this set can be much larger. In order to obtain more reliable results, the experiment should be repeated with larger sets of requirements.
- **Selection of Bias:** We have supplied a set of 100 requirements in order to evaluate automatic validation. Although a significant effort is spent to make the requirements as homogeneous and realistic as possible, the distribution of the requirement types played a significant role to obtain the metrics. With another requirement set, the obtained metrics can be different depending on the information presented in the requirements.

- **Changes on the Requirements:** The recall of the automatic validation for the unstructured requirements strictly depends on the which part of the requirement changes. We have requested an engineer to change the requirements. However, in reality, requirements do not change randomly and there should be a reason behind the change. For more accuracy, we need real requirements documents with their history, in order to determine which type of requirements change and what part of the requirement changes. Alternatively, a mission design process can be observed entirely and more reliable metrics can be obtained through the requirement changes during the process.

However, to address the threats mentioned above, we put the main focus not on the pure metrics, but the reasons and mechanism responsible for creating those metrics. In another experiment with a different set of requirements, although the metrics would be different, the same deductions can be made about factors which affect the automatic validation.

- **Gathered Data for Requirement-Based Artifact Reuse:** For the quantitative evaluation, we have gathered data from a small set of participants. Furthermore, the participants have different backgrounds and skills, which may have affected the outcome. In order to provide exact metrics, we need a much larger set of participants with similar background and skills. We address this threat by not drawing statistical conclusions from the experiment. We simply give an overview, which shows that, accuracy of finding reusable artifacts is increased and the time spent on finding reusable artifacts are decreased with our contribution.

External Validity

We have demonstrated the capabilities of our methodology and implementation by integrating ReqTrace with Virtual Satellite. Our evaluation results for automatic validation rate, artifact generation and requirement-based artifact reusability are strictly affected by the spacecraft model and the functionalities of the Virtual Satellite. As an example, with another software, we could get much higher or lower automatic validation rate depending on the information modeled. To ensure external validity, we need to repeat our evaluation with different modeling tools. However, the main focus of our evaluation is not about showing the capabilities of ReqTrace. Rather, we have proved that our methodology can provide the benefits we have claimed, when it is properly implemented. Other modeling tools may not be integrated with ReqTrace, however, different tools can be developed regarding our theoretical contribution by using different technologies.

6 Related Work

We will review the related literature in this chapter. We focus on requirements engineering and we briefly present the related research conducted on requirements traceability. Moreover, as we have discussed in our methodology, extracting information from requirements is challenging since most of the time requirements are given in natural language. To overcome this issue, we have proposed using requirement attributes to present the important information, and defined validation engines to specify what to do with those information. However, different techniques for extracting information from requirements also exist. We give an overview on other techniques to extract information from requirements.

Contributions on Traceability

Gotel et al. [32] analyzed the current tool support for requirements traceability and the challenges of requirements traceability. In their study, they have interviewed over 100 practitioners and analyzed the current tool support for requirements traceability. They have introduced the terms *pre-requirements specification traceability* and *post-requirements specification traceability*. Since our methodology is concerned with tracing requirements to the model artifacts, it falls under *post-requirements specification traceability*. However, they have stated that *pre-requirements specification traceability* is also important for the success of a project. Their conclusion was that, existing tool support for *pre-requirements specification traceability* is not adequate. But on the contrary, scientific research and commercial tool support mostly focuses on *post-requirements specification traceability*. Their findings are important to lead the research community, since both of the traceability classifications play an important role regarding the success of a project.

Winkler and Pilgrim analyzed the state of art of traceability in requirements engineering and model-driven development [51]. In their paper, they first have analyzed the history of the concept of traceability, the need for traceability and the basic principles of traceability. They have surveyed the existing approaches regarding the creation of traceability links such as automatic creation with language processing, maintenance and classification. They have concluded that, traceability in requirements engineering is used far more often than model-driven development. However, in both fields, the terminology is not mature, and there is a lack of research on global level. In other words, researches are conducted on specific domains and they use different terminology depending on the domain. They have also concluded that, in both fields the existing tool support for maintenance of the traceability is not adequate. Our work can be considered as a way to fill this gap.

Galvao and Goknil [13] classify three different traceability categories related with the model driven engineering: requirements-driven approaches, which is tracing the requirements to model artifacts. Modeling approaches, which is tracing model artifacts to different model artifacts in order to capture their relation. Transformation approaches, which is tracing the artifacts in the original model to the transformed model. They survey five different approaches regarding requirements-driven approaches, four from modeling and three from transformation. They then analyze those approaches

regarding four criteria defined by them which are mapping (can the approach support mapping in different types of artifacts), scalability, change impact analysis and tool support. Our work would fall under the requirements-driven approaches, and we provide the change impact analysis and tool support. We only support inter mapping, tracing the requirements to model artifacts.

Tufail et al. [52] analyzed the challenges and new developments in the area of requirements traceability during 2010 and 2017. They have found seven different models for traceability, ten different challenges, and fourteen tools. They have analyzed the traceability models and requirement management tools regarding their criteria. Similar with our criteria for existing tools, their criteria also include change impact analysis and different types of traceability links. They have concluded that, DOORS is the best requirement management tool and Traceability Meta Model is the best metamodel. However, as analyzed in Subsection 2.4.1, even DOORS do not provide the features of our work.

Application of Traceability

Having complete traceability infromation in a project is desired, however maintaining and creating those traceability links is expensive. For that reason in many projects, traceability information is not kept. There has been some research conducted on to determine the factors on the practical usage of traceability. Bouillon et al. [53] conducted a comprehensive research on the usage of traceability. They have first identified 29 different use cases for the application of traceability from the existing literature, such as tracing the requirements to the test cases, tracing the requirements to design artifacts, tracing the requirements to budget estimations. Then, they have gathered data from 117 developers regarding the practical usage of the traceability in each use case. Then regarding the answers from the participants. They have found that, traceability is used mostly in two scenarios. First one is to determine the requirement origin (Pre-Requirements Traceability) and second one is to trace the requirements to the code. According to their findings, tracing requirements to the design artifacts is not common in practice. With our methodology, we have addressed this issue, and demonstrated the benefits of tracing the requirements to models.

Mäder et al. [54] conducted an experiment with 71 developers in order to find the advantages of traceability. First some changes made on the requirements document or use cases. Then a group of developers were given the tasks to update the existing code regarding the changes. Half of the tasks had traceability to code, whereas the other half did not. Authors analyzed the correctness of the solutions as well as performance. The experiment showed that with traceability information developers produced 50% more correct solutions. Moreover, developers solved the tasks with 24% percent faster with traceability information.

Watkins and Neal [55] analyze the advantages of traceability in defense projects. In their paper, they categorize four aspects where traceability information significantly helps the developers. These are; verification& validation, cost reduction, change management and accountability. To further benefit from traceability information, they propose to improve the way the requirements are written so that better requirements would create simpler and more concrete traceability links. In our methodology, we have addressed this issue by providing guidelines for space mission requirements and templates for structured requirements, so that they can be traced and validated reliably.

Traceability Models

In our thesis, we have provided a relatively simple traceability model. Our model only supports traceability links from requirements to model artifacts. There has been some research to create a universal generic traceability model to be used in both requirements engineering and model based development for model transformations.

Mustafa et al. [56] argue the concept of generic traceability model in their paper. According to their research; the generic traceability model should allow; modeling traceability between artifacts of same or different types, modeling traceability between source and target artifacts of one to one, one to many and many to many cardinalities, specifying the direction of the traceability links, capturing traceability information within the model or across different models, putting constraints on trace elements, modeling traceability between elements of different granularity, the generic model should prohibit establishing illegal traceability links and the model should be flexible to insert new types of traceability links without changing the model. In their research, they have not found a generic traceability model to meet all the requirements. However, if such a generic traceability model is provided, our methodology can benefit from it. Our traceability model can be replaced or merged with the generic traceability model to improve its functionality.

Pattern Based Requirements Writing

We have provided a set of patters in order to write structured requirements for the space missions in our methodology. This set of patterns are also used in other domains. Another usage of limited language in a different domain is called The Requirement Specification Language [57]. Holtmann et al. proposes using a Controlled Natural Languge (CNL) for functional system requirements in the automotive domain. In their approach, requirements written with natural language first translated in to CNL manually. With this process, synonyms and ambiguities in the natural language are removed. One example from their technique is;

Customer requirement:

- *A comfort control unit has to be created that interconnects the functionalities of the central locking system, interior light control, … .*

Requirement rewritten with CNL :

- The system “Comfort Control Unit” consists of the following subsystems: **Central Locking**, **Interior Light Control**, … .
- The system “Interior Light Control” processes the following signals: **Doors Unlocked**, …

For the safety critical systems, Teemu et Al. [58] created a comprehensive structured grammar to cover every requirement. Their motivation was also to eliminate the ambiguities and provide more suitable requirements for automatic analysis. One example from their work is as follows;

- The <system function> shall be able to <action> <entry>

The parts in the angle brackets represents the variables. A concrete example requirement fits in this boilerplate can be;

- The *missile launcher* shall be able to *launch missiles.*

This requirement is now suitable for automated analysis while the terms can easily be extracted. However, this approach is useful if a glossary for the requirements specification document is defined. In the glossary if the *missile launcher* is defined as a system function then it can be analyzed automatically by computers.

When such boilerplates are required to be used in a project, existing requirements needs to be checked to determine whether they are conforming to the boilerplate or not. When done manually, this is a time consuming job for the developers. The process can be fully automated if a glossary for the requirements specification document is defined. Glossaries contain the definition of each term used in the requirements. In the glossary, if the *missile launcher* is defined as a system function, then it is possible to automatically confirm the conformance. However, every requirement specification document does not contain those glossaries. To address this issue, Arora et al.[59] developed a methodology using a natural language processing technique called *text chunking* to detect the conformance of the requirements to the given boilerplate when the requirements specification document does not contain a glossary. With their approach, they were able get a precision of 92% and a recall of 96%.

Glossaries can be defined for individual requirement documents as well as for different domains. But also there has been some research for creating a global glossary. The idea of a global glossary is to fix the terms to be used in requirements, independent of the domain. Martin Glinz [60] from international requirements engineering board published a global glossary with 128 terms and their definitions.

Natural Language Processing

In addition to using structured language and templates to extract information, natual language processing is another technique to extract information. Zisman et al. [19]investigated natural language processing to automatically parse requirements and create trace relations with other software artifacts like use cases. In their approach, the requirements are first parsed automatically, then the important terms are tagged, then a traceability relation between the requirements from different sets are established. To evaluate their approach, they have selected two sets of specifications for a television product family, namely commercial requirements specification and functional requirements specification. They evaluated their work by comparing auto generated trace relations with a set of trace relations defined by three experts. On average, they were able to create the 80% of the traceability information automatically.

Sclutter and Vogelsang [20] described a natural language processing pipeline, where first requirements are normalized, then using a technique called Semantic Role Labeling, information triplets are extracted from the requirements. Each triplet contains information of object, subject and relation. They have then merged all of the triplets gathered from the requirements, to create a knowledge graph. With their approach it is possible to analyze the relations among requirements. Their approach has some drawbacks such as; they cannot understand the meaning of the terms. Terms such as "vehicle" and "car" are considered two different objects, however they are the same object. Secondly, their approach cannot handle conditional requirements.

Since natural language processing is not mature enough, we did not perform this technique to extract information from the requirements.

Ontology Extraction

Another technique in requirements engineering for both removing the ambiguities in the requirements and aiding automatic requirement parsing is the usage of ontologies. Tom Gruber [61] defines an ontology as *a specification of a conceptualization.* Ontologies are more advanced than glossaries. In glossaries, only the meaning of each term is depicted. However, with ontologies relationships between different terms are also captured. Ontologies provide a number of benefits. First, requirements written regarding ontologies will not contain ambiguity or synonyms. Second, since all the terms are defined, computers can perform the textual matching and analysis of the requirements automatically.

These onthologies can be predefined by human experts as well as they can be extracted from a given requirements specification document. Ben Achour [62] classifies the existing term extraction techniques to three categories; lexical, syntactic and semantic.

Leonid kof [63], in his paper, explains the three techniques as; Lexical approaches consider a sentence as a character sequence. If a character sequence occurs multiple times in a document, it is considered as a term. Syntactical approaches use the grammar of a sentence to extract the terms. They can provide more information when compared to lexical approaches, however, that technique demands grammatically correct sentences. Semantical approaches on the other hand, interpret each sentence as a logical formula. Since they demand a firm sentence structure, this technique is not applicable to many real world requirements written in natural language.

7 Conclusion

In this thesis, we have created a new methodology for requirements management. Although our methodology is applicable for different systems engineering domains, we have focused on space mission requirements and spacecraft models in particular. In the first part of the methodology, traceability between the requirements and model artifacts are covered. Unlike the existing traceability approaches, our traceability methodology provides automatic validation. Furthermore, we have analyzed the structured requirements, and provided the boilerplates for the requirements which are suitable for automatic validation. Moreover, we have explained how to generate these structured requirements automatically. With these structures, requirements can be more reliably traced. Furthermore, such structured requirements provide advantages for requirements sharing, since they can be more efficiently validated. One important aspect is that, automatic validation is achievable if the requirements are traced to relevant artifacts, however, requirements can be traced to irrelevant artifacts due to human mistake. For that reason, in the second step of our methodology, we have introduced modeling on the basis of requirements. When the model artifacts are created on the basis of the requirements, this room for human mistake is eliminated. Moreover, we have provided different use cases where artifact generation on the basis of requirements can be beneficial for the engineers.In the final step of our methodology, we have analyzed requirement-based reusability of model artifacts. Instead of manually detecting the resuability of the existing design, reusable artifacts from the previous projects can be automatically detected with our automatic validation mechanism. When this detection performed automatically, these artifacts can be found faster and more reliable.

In order to demonstrate the capabilities of our methodology, we have developed a framework called ReqTrace. The framework provides the functionalites we have described in our methodology. We have evaluated our implementation by integrating it with the Virtual Satellite, a software for modeling a spacecraft. We have used a conceptual spacecraft model called KatSat which is modeled with Virtual Satellite and 100 requirements about different components of the spacecraft. From the evaluation, we have observed that, approximately half of the requirements could be traced with our automatic validation mechanism. Furthermore, when the requirements are changed, 65% of the violations are automatically detected and all of the detected violations were true violations. This would give 100% precision and 65% recall. The main cause of missing the violations is that, unstructured requirements are vulnerable against the changes made on the *Action* part. When the same evaluation with the structured requirements is repeated, we were able to obtain 100% precision and 100% recall. In other words, every violation is automatically detected and every detected violation is a true violation. For the artifact generation, we have performed a qualitative analysis and demonstrated which artifacts can be created depending on the initial requirements as well as how artifact generation can be used to modify the model to make it compatible with the changes on the requirements. For the requirements-based reusability of the model artifacts, first, we have performed a qualitative analysis to give an overview about how the functionality works. Furthermore,

we have performed a quantitative analysis with a set of participants, which showed that, users can find suitable artifacts more reliably and faster when compared to manual detection.

In conclusion, we have realized our goals given in Chapter 1. The evaluation showed that, our methodology improves the state of art requirements management. All three steps of the methodology produced the desired results. Moreover, the contribution of this thesis is not purely theoretical, since it has been shown that, our implementation can be integrated with a real spacecraft modeling tool, and can be used in real life modeling applications.

8 Future Work

The methodology we have presented in this thesis gives several directions for the future work. In this chapter, we explain some of them briefly.

Comprehensive Analysis of Space Mission Requirements

Due to the time constraints, and the confidentiality of the space missions, we have analyzed requirements about several different space missions to base on our methodology. In order to improve the capabilities of our methodology and implementation, we plan to investigate more requirements for space missions, since different requirements may require different strategies for the automatic validation. However, obtaining access to confidential documents is still a challenge for us.

Evaluation on other System Engineering Domains

We have demonstrated our methodology on the basis of space mission requirements and spacecraft models. However, automatic validation, artifact generation and reusability of the requirements and models, are not only beneficial in space domain. In other system engineering domains, such as automotive engineering and marine engineering, our methodology can provide the same benefits. We believe that, there should be sufficient amount of requirements which can be automatically validated in those domains as well. However, in order to prove our claim, we need to evaluate our methodology with different modeling tools from different disciplines.

Tracing the Requirements to Simulation Results

In our methodology we have targeted the model artifacts to trace the requirements for automatic validation. One other idea is to trace the requirements to the simulations. Many requirements in the aerospace domain is about different faults happening in the spacecraft. Consider the following requirement;

- The spacecraft shall be capable to receive telecommands and transmit telemetry in any attitude and position during a ground station pass.

This requirement cannot be validated automatically just by tracing it to a model artifact since the model artifact would not contain enough data. However, simulation results can provide the necessary data to validate the requirement.

Tool Support with Graphical Editor

In requirements engineering, tool support is considered as important as the methodology. Currently we can trace and automatically validate the requirements. However, we do not have a good editor to observe the links between requirements and model artifacts. It would be useful for the users to observe the requirements and the model artifacts in a big graph.

Expanding the Methodology for the Requirements Coverage Analysis

In our methodology, we have mainly used our traceability approach to detect the changes on the model artifacts and requirements. However, traceability between requirements and model artifacts also ensures that each requirement is satisfied by some artifact and each artifact exists in order to satisfy a requirement. It is also possible to analyze the coverage of the requirements in order to determine which of them are traced and which of them are not. With that data it would be possible to estimate the time needed for finishing the project and cost of the project. Similarly, we can also analyze the model artifacts, which of them are related with a requirement and which of them are not. With this expansion we could determine the redundantly implemented artifacts and avoid further investments in those artifacts.

Pre-Requirements Traceability

In our methodology, we have analyzed the impact of the requirement changes on the model artifacts. However, changes on a requirement may require other related requirements to change. As an example, if requirements related with the total mass of the spacecraft changes, then the requirements related with the mass of the individual components should also change. Our methodology can be used in this step to detect the impact of a change on the other requirements. Furthermore, if such a change occurs, we could automatically analyze the related requirements and automatically validate them. Moreover, we could use our artifact generation technique to create requirements on the basis of other requirements. As an example, on the basis of the total mass requirement of the spacecraft, different requirements can be generated for the mass of the different components.

Bibliography

[1] "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (12/1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[2] B. Nuseibeh and S. Easterbrook. "Requirements Engineering: A Roadmap". In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. Limerick, Ireland: ACM, 2000, pp. 35–46. ISBN: 1-58113-253-0. DOI: 10.1145/336512.336523. URL: http://doi.acm.org/10.1145/336512.336523.

[3] "IEEE Standard for System and Software Verification and Validation". In: *IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004)* (05/2012), pp. 1–223. DOI: 10.1109/IEEESTD.2012.6204026.

[4] T. S. G. Internationa. *The CHAOS Manifesto, 2013: Think Big, Act Small*. 2013.

[5] A. Mandal and S. C Pal. "Identifying the Reasons for Software Project Failure and Some of their Proposed Remedial through BRIDGE Process Models". In: 3 (01/2015), pp. 118–126.

[6] B. W. Boehm and P. N. Papaccio. "Understanding and Controlling Software Costs". In: *IEEE Trans. Softw. Eng.* 14.10 (10/1988), pp. 1462–1477. ISSN: 0098-5589. DOI: 10.1109/32.6191. URL: http://dx.doi.org/10.1109/32.6191.

[7] D. M. Fernández, S. Ognawala, S. Wagner, and M. Daneva. "Where Do We Stand in Requirements Engineering Improvement Today?: First Results from a Mapping Study". In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '14. Torino, Italy: ACM, 2014, 58:1–58:4. ISBN: 978-1-4503-2774-9. DOI: 10.1145/2652524.2652555. URL: http://doi.acm.org/10.1145/2652524.2652555.

[8] D. Kavitha and A. Sheshasaayee. "Requirements Volatility in Software Maintenance". In: *Advances in Computer Science and Information Technology. Computer Science and Information Technology*. Ed. by N. Meghanathan, N. Chaki, and D. Nagamalai. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 142–150. ISBN: 978-3-642-27317-9.

[9] D. Zowghi and N. Nurmuliani. "A study of the impact of requirements volatility on software project performance". In: *Ninth Asia-Pacific Software Engineering Conference, 2002.* (). DOI: 10.1109/apsec.2002.1182970.

[10] B. Curtis, H. Krasner, and N. Iscoe. "A field study of the software design process for large systems". In: *Communications of the ACM* 31.11 (01/1988), pp. 1268–1287. DOI: 10.1145/50087.50089.

[11] S. Nair, J. L. D. L. Vara, and S. Sen. "A review of traceability research at the requirements engineering conferencere@21". In: *2013 21st IEEE International Requirements Engineering Conference (RE)* (2013). DOI: 10.1109/re.2013.6636722.

[12] D. C. Schmidt. "Model-Driven Engineering". In: *IEEE computer* (2006). DOI: doi.ieeecomputersociety.org/10.1109/MC.2006.58. URL: https://www.fing.edu.uy/inco/grupos/coal/uploads/Main/mdepaper.pdf.

[13] I. Galvao and A. Goknil. "Survey of Traceability Approaches in Model-Driven Engineering". In: *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*. 10/2007, pp. 313–313. DOI: 10.1109/EDOC.2007.42.

[14] "IEEE Standard for Software Verification and Validation". In: *IEEE Std 1012-1998* (07/1998), pp. 1–80. DOI: 10.1109/IEEESTD.1998.87820.

[15] IEEE. "Systems and software engineering – Life cycle processes –Requirements engineering". In: *ISO/IEC/IEEE 29148:2011(E)* (2011).

[16] N. Boulila. *Guidelines for Good Requirements Writing with Examples*. 05/2014. DOI: 10.13140/RG.2.1.5135.8168.

[17] I. Hooks. "Writing Good Requirements". In: *INCOSE International Symposium* 4.1 (1994), pp. 1247–1253. DOI: 10.1002/j.2334-5837.1994.tb01834.x.

[18] *Writing Good Requirements: Checklists*. 06/2018. URL: https://ep.jhu.edu/about-us/news-and-media/writing-good-requirements-checklists.

[19] A. Zisman, G. Spanoudakis, E. Pérez-Miñana, and P. Krause. "Tracing Software Requirements Artefacts". In: 1 (01/2003), pp. 448–455.

[20] A. Schlutter and A. Vogelsang. "Knowledge Representation of Requirements Documents Using Natural Language Processing". In: ().

[21] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak. "Easy approach to requirements syntax (EARS)". In: 10/2009, pp. 317–322. DOI: 10.1109/RE.2009.9.

[22] M. Warnier and A. Condamines. "Improving Requirement Boilerplates Using Sequential Pattern Mining". In: *Europhras 2017*. Londres, United Kingdom, 11/2017. URL: https://hal.archives-ouvertes.fr/hal-01672313.

[23] B. W. Boehm. *Software Engineering Economics*. Upper Saddle River, NJ, USA, 1981.

[24] L.Sharma. *Difference between Verification and Validation*. URL: http://toolsqa.com/software-testing/difference-between-verification-and-validation/.

[25] I. R. E. Board. *Open Up: How the ReqIF Standard for Requirements Exchange Disrupts the... - Requirements Engineering Magazine*. URL: https://re-magazine.ireb.org/issues/2014-3-gaining-height/open-up/.

[26] URL: https://www.omg.org/spec/ReqIF/About-ReqIF/.

[27] *Technical Aspects*. URL: https://reqif.de/en/resources/articles/tech-aspects.html.

[28] C. V. Ramamoorthy, V. Garg, and A. Prakash. "Programming in the large". In: *IEEE Transactions on Software Engineering* SE-12.7 (07/1986), pp. 769–783. ISSN: 0098-5589. DOI: 10.1109/TSE.1986.6312978.

[29] G.-C. Roman. "A taxonomy of current issues in requirements engineering". In: *Computer* 18 (1985), pp. 14–23.

[30] P. Lago, H. Muccini, and H. van Vliet. "A Scoped Approach to Traceability Management". English. In: *Journal of Systems and Software* 82.1 (2009), pp. 168–182. ISSN: 0164-1212. DOI: 10.1016/j.jss.2008.08.026.

[31] N. Anquetil, B. Grammel, I. Galvao, J. Noppen, S. Shakil Khan, H. Arboleda, A. Rashid, and A. Garcia. "Traceability for Model Driven, Software Product Line Engineering". Undefined. In: *ECMDA Traceability Workshop Proceedings*. Supplement. SINTEF, 06/2008, pp. 77–86. ISBN: 978-82-14-04396-9.

[32] O. Gotel and A. Finkelstein. "An analysis of the requirements traceability problem". In: *ICRE*. 1994.

[33] G. Shea. *NASA Systems Engineering Handbook Revision 2*. 06/2017. URL: https://www.nasa.gov/connect/ebooks/nasa-systems-engineering-handbook/.

[34] *Home*. URL: http://pericles-project.eu/training-module/space-data/space-project-phasing-data-levels-and-data-use/project-phases-esa-project/.

[35] J. R. Wertz, W. Larson, and B. D'Souza. *SMAD III: Space mission analysis and design*. Microcosm Press, 2005.

[36] M. Kassab, O. Ormanjieva, and M. Daneva. "A Traceability Metamodel for Change Management of Non-functional Requirements". Undefined. In: *Proceedings of the 6th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2008*. Ed. by W. Dosch, R. Lee, P. Tuma, and T. Coupaye. WoTUG-31. 10.1109/SERA.2008.37. United States: IEEE Computer Society, 08/2008, pp. 245–254. ISBN: 978-0-7695-3302-5. DOI: 10.1109/SERA.2008.37.

[37] C. Lange, J. T. Grundmann, M. Kretzenbacher, and P. M. Fischer. "Systematic reuse and platforming: Application examples for enhancing reuse with model-based systems engineering methods in space systems development". In: *Concurrent Engineering* 26.1 (2018), pp. 77–92. DOI: 10.1177/1063293X17736358. eprint: https://doi.org/10.1177/1063293X17736358. URL: https://doi.org/10.1177/1063293X17736358.

[38] D. Di Domizio and P. Gaudenzi. "A Model for Preliminary Design Procedures of Satellite Systems". In: *Concurrent Engineering* 16.2 (06/2008), pp. 149–159. DOI: 10.1177/1063293X08092488. URL: https://hal.archives-ouvertes.fr/hal-00571221.

[39] J. R. Wertz, D. F. Everett, and J. J. Puschell. *Space mission engineering: the new SMAD*. Microcosm Press, 2015.

[40] *S2TEP*. URL: https://www.dlr.de/irs/en/desktopdefault.aspx/tabid-12525/21846_read-49985/.

[41] E. Kindler. "Model-based software engineering: The challenges of modelling behaviour". In: (01/2010), p. 4.

[42] R. Gronback. *Eclipse Modeling Project | The Eclipse Foundation*. URL: https://www.eclipse.org/modeling/emf/.

[43] P. M. Fischer. "Enabling a Conceptual Data Model and Workflow Integration Environment for Concurrent Launch Vehicle Analysis". In: *69th International Astronautical Congress (IAC)* ().

[44] *Virtual Satellite*. URL: https://www.dlr.de/sc/en/desktopdefault.aspx/tabid-5135/8645_read-8374/.

[45] *45 Best Requirements Management Tools (The Complete List)*. 08/2018. URL: https://www.softwaretestingh
com/requirements-management-tools/.

[46] *IBM Rational DOORS*. URL: https://discovery.hgdata.com/product/ibm-rational-doors.

[47] M. Jastram. *Eclipse Requirements Modeling Framework*. URL: https://www.eclipse.org/rmf/.

[48] J.-P. Steghöfer. *Capra*. 07/2016. URL: https://projects.eclipse.org/proposals/capra.

[49] C. Guindon. *ReqCycle | The Eclipse Foundation*. URL: https://www.eclipse.org/proposals/polarsys.reqcycle/.

[50] *Star Trackers*. URL: https://www.terma.com/space/space-segment/star-trackers/.

[51] S. Winkler and J. Pilgrim. "A Survey of Traceability in Requirements Engineering and Model-driven Development". In: *Softw. Syst. Model.* 9.4 (09/2010), pp. 529–565. ISSN: 1619-1366. DOI: 10.1007/s10270-009-0145-0. URL: http://dx.doi.org/10.1007/s10270-009-0145-0.

[52] H. Tufail, M. F. Masood, B. Zeb, F. Azam, and M. W. Anwar. "A systematic review of requirement traceability techniques and tools". In: *2017 2nd International Conference on System Reliability and Safety (ICSRS)*. 12/2017, pp. 450–454. DOI: 10.1109/ICSRS.2017.8272863.

[53] E. Bouillon, P. Mäder, and I. Philippow. "A Survey on Usage Scenarios for Requirements Traceability in Practice". In: *Requirements Engineering: Foundation for Software Quality*. Ed. by J. Doerr and A. L. Opdahl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 158–173. ISBN: 978-3-642-37422-7.

[54] P. Mäder and A. Egyed. "Do developers benefit from requirements traceability when evolving and maintaining a software system?" In: 20 (04/2014), pp. 1–29.

[55] R. Watkins and M. Neal. "Why and how of requirements tracing". In: *IEEE Software* 11.4 (07/1994), pp. 104–106. ISSN: 0740-7459. DOI: 10.1109/52.300100.

[56] N. Mustafa and Y. Labiche. "Towards traceability modeling for the engineering of heterogeneous systems". In: *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 02/2015, pp. 321–328.

[57] J. Holtmann, J. Meyer, and M. Meyer. "A Seamless Model-Based Development Process for Automotive Systems". In: (02/2011), pp. 79–88.

[58] T. Tommila and A. Pakonen. *Controlled natural language requirements in the design and analysis of safety critical I&C systems*. English. Research report. Project code: 77380 / SAREMAN-2013. Finland: VTT Technical Research Centre of Finland, 2014.

[59] C. Arora, M. Sabetzadeh, L. C. Briand, and F. Zimmer. "Requirement boilerplates: Transition from manually-enforced to automatically-verifiable natural language patterns". In: *2014 IEEE 4th International Workshop on Requirements Patterns (RePa)*. 08/2014, pp. 1–8. DOI: 10.1109/RePa.2014.6894837.

[60] M.Glinz. "A Glossary of requirements engineering terminology". In: *International Requirements Engineering Board* (2011).

[61] T. Gruber. URL: http://www-ksl.stanford.edu/kst/what-is-an-ontology.html.

[62] B. Achour. "Linguistic Instruments for the Integration of Scenarios in Requirement Engineering". In: *Proceedings of the Third International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'97)* (1997).

[63] L. Kof. “Natural Language Processing: Mature Enough for Requirements Documents Analysis?” In: *Natural Language Processing and Information Systems*. Ed. by A. Montoyo, R. Muńoz, and E. Métais. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 91–102. ISBN: 978-3-540-32110-1.

A Requirements

In this chapter, we present the requirements which we have used to evaluate our methodology.

A.1 Requirements for the Spacecraft Model

Table A.1: Numerical Type Requirements

ID	Description	Value	Unit
SAT-RW-001	The mass of the reaction wheel should not exceed	300.0	Gram
SAT-RW-002	The average power consumption of the reaction wheel should be less than	8.0	Watt
SAT-RW-003	The volume of the reaction wheel should be at most	105.0	cm^3
SAT-STR-001	Power consumption of the star tracker should be less than	2.0	Watt
SAT-STR-002	The acquisition time of the star tracker should be less than	1.0	Second
SAT-STR-003	Update rate of the star tracker should not be less than	10.0	Hertz
SAT-STR-003.1	Optionally update rate should not be less than	45.0	Hertz
SAT-STR-004	Input voltage of the star tracker should be	5.0	Volt
SAT-STR-005	Lifetime of the of the star tracker should not be less than	3.0	Year
SAT-STR-006	The weight of the star tracker should be at most	250.0	Gram
SAT-SunSen-001	Power consumption of the sun sensor should be less than	2.0	Watt
SAT-SunSen-002	The weight of the sun sensor should not be more than	50.0	Gram
SAT-SunSen-003	The field view of the sun sensor should be more than	130.0	Degree
SAT-SunSen-004	The data update rate of the sun sensor should be more than	4.0	Hertz
SAT-SunSen-005	The sun sensor should be supplied with the following voltage	5.5	Volt
SAT-GPSREC-001	The gps receiver should provide correct information in the speeds up to	9.0	Km/hour

Continued on next page

Table A.1 – continued from previous page

ID	Description	Value	Unit
SAT-GPSREC-002	The position update rate of the receiver should be at least	1.0	Hertz
SAT-GPSREC-003	Operating temperature of the receiver should not exceed	90.0	Celcius
SAT-GPSREC-004	The acquisition sensitivity of the receiver should be at least	30.0	dBc/Hertz
SAT-GPSREC-005	Power consumption of the receiver should not be more than	0.5	Watt
SAT-GPSREC-006	The weight of the receiver should not exceed	50.0	Gram
SAT-GPSANT-001	Power consumption of the gps antenna should be less than	1.0	Watt
SAT-GPSANT-002	Input voltage of the gps antenna should be	3.3	Volt
SAT-GPSANT-003	The gps antenna should work with temperatures up untill	100.0	Celcius
SAT-GPSANT-004	The weight of the gps antenna should not exceed	55.0	Gram
SAT-DDS-001	The bandwitdth of the DDS antenna should be	10.0	MHertz
SAT-DDS-002	Lifetime of the DDS antenna should be at least	5.0	Year
SAT-DDS-003	Operating temperature of the DDS antenna should not exceed	110.0	Celcius
SAT-DDS-004	DDS Antenna gain in boresight should not be less than	6.0	dbi
SAT-DDS-005	The weight of the DDS antenna should not exceed	85.0	Gram
SAT-TTC-001	The bandwitdth of the TTC antenna should be	300.0	MHertz
SAT-TTC-002	The impedance of the TTC antenna should be at least	40.0	Ohm
SAT-TTC-003	Operating temperature of the TTC antenna should not exceed	90.0	Celcius
SAT-TTC-004	TTC Antenna gain should not be less than	2.0	dbi
SAT-TTC-005	The weight of the TTC antenna should not exceed	150.0	Gram
SAT-OBC-001	The RS422 UART interface of the OBC shall have either one of the following bitrates:	0.0	
SAT-OBC-001.1	lower option	19200.0	bps
SAT-OBC-001.2	higher option	115200.0	bps
SAT-OBC-002	Power consumption of the OBC should be less than	10.0	Watt
SAT-OBC-003	The weight of the OBC should not exceed	2500.0	Gram

Continued on next page

Table A.1 – continued from previous page

ID	Description	Value	Unit
SAT-OBC-004	The on-board computer shall provide a non-volatile data storage capacity equal to or greater than	260.0	Mbyte
SAT-PCDU-001	The temperature of the PCDU should not exceed	100.0	Celcius
SAT-PCDU-002	The PCDU should be supplied with	3.3	Volt
SAT-PCDU-003	The mass of the PCDU should not exceed	4500.0	Gram
SAT-BAT-001	The battery should provide the following voltage	3.7	Volt
SAT-BAT-002	The weight of the battery should not exceed	150.0	Gram
SAT-IRCAM-001	Power consumption of the camera should be less than	5.0	Watt
SAT-IRCAM-002	The weight of the camera should not be more than	3500.0	Gram
SAT-IRCAM-003	Camera linerate should be above	9.0	Hertz
SAT-IRCAM-004	Pixel rate of the camera should be above	20.0	Mpixel/sec
SAT-IRCAM-005	The camera should be supplied with the following voltage	12.0	Volt
SAT-IRCAM-006	The camera should operate in temperatures up to	85.0	Celcius

Table A.2: Enum Type Requirements

ID	Description	Enum
SAT-SYS-001	The spacecraft should have the following system modes	safemode, idlemode, nadir-pointingmode, Detumbling, firedetection
SAT-RW-004	ReactionWheel should provide one of the following interfaces	ITC, UART, CANBus
SAT-RW-005	ReactionWheel should not be active during	safeMode
SAT-RW-006	ReactionWheel should be active during	firedetection
SAT-STR-007	StarTracker should not be active during	Detumbling, safeMode, DHO
SAT-STR-008	StarTracker should be active during	FireDetection, NadirPointing-Mode
SAT-STR-009	StarTracker should be connected through the following interfaces	CANBus
SAT-SunSen-006	The sensor should be connected thorugh the following interface	RS485

Continued on next page

Table A.2 – continued from previous page

ID	Description	Enum
SAT-SunSen-007	The sun sensor should be active during the following modes	FireDetection, Detumbling, SafeMode
SAT-SunSen-008	The sun sensor should not be active during the following modes	DHO, NadirPointingMode
SAT-GPSREC-007	The receiver should provide the interface of type	UART
SAT-GPSREC-008	The GPS receiver should support the following protocols	pNAV, NMEA0183
SAT-GPSREC-009	The GPS receiver should not be active during	Detumbling, safeMode, DHO
SAT-GPSREC-010	The GPS receiver should be active during	FireDetection, NadirPointingMode
SAT-GPSANT-005	The GPS antenna should provide the following interface	UART
SAT-GPSANT-006	The GPS antenna should not be active during	Detumbling, safeMode, DHO
SAT-GPSANT-007	The GPS antenna should be active during	FireDetection, NadirPointingMode
SAT-DDS-006	The DDS antenna should provide the interface of type	SMA
SAT-TTC-007	The TTC antenna should provide the interface of type	SMA
SAT-TTC-008	The TTC antenna should be active during the following modes	safemode, idlemode, nadirpointingmode, Detumbling, firedetection
SAT-TTC-009	The TTC antenna should be passive during the following modes	DHO
SAT-OBC-005	All data interfaces connected to the OBC shall be selected from the following types of interfaces:	UART, I2C, RS485
SAT-OBC-006	The OBC should have the following modes	DHO, OperationMode, FDIR
SAT-PCDU-004	PCDU should provide the following interfaces	UART, SMA, RS422
SAT-PCDU-005	PCDU should be active during	DHO
SAT-IRCAM-007	The camera should be connected thorugh the following interfaces	I2C, CANBus
SAT-IRCAM-008	The camera should be active during the following modes	FireDetection, OperationMode

Table A.3: General Type Requirements

ID	Description
SAT-SYS-001	The spacecraft should detect forest fires with infrared technology
SAT-SunSen-009	The sun sensor should accuretaly determine the position of sun.
SAT-SunSen-010	The sun sensor should have a reflective metal coating
SAT-GPSREC-011	The receiver should provide non stop position measurement
SAT-GPSREC-012	The receiver should have protection agains radiation
SAT-GPSREC-013	The receiver should support active or passive GPS antennas
SAT-GPSANT-008	The Gps antenna should have proper insulation.
SAT-DDS-007	The DDS antenna should provide either RH or LH circular polarization
SAT-DDS-008	The DDS antenna should be radiation tolerant
SAT-TTC-009	The TTC antenna should provide RH and LH circular polarization
SAT-TTC-010	The TTC antenna should have a protective cover
SAT-OBC-007	The parity bit of the RS422 UART interface of the OBC shall be configurable to no, odd or even parity.
SAT-OBC-008	The parity bit of the RS422 UART interface of the OBC shall be disabled by default.
SAT-OBC-009	The on-board computer shall provide the ability to execute software applications.
SAT-BAT-004	Battery should have minimum of 2 years radiation tolerance
SAT-IRCAM-009	The camera should be controllable from a pc
SAT-IRCAM-010	The camera should be able to work in low light conditions
SAT-HAR-001	The harness shall interconnect all subsystems and components of the spacecraft for the transmission of power and analog as well as digital signals.
SAT-HAR-002	All harnesses and cables shall be protected against abrasion, cold flow, cut through, vibration, chafing, flexing and damage by sharp edges.
SAT-HAR-003	All connectors shall be labeled with a unique designator.
SAT-HAR-004	Power and data lines shall be routed through separated connectors.
SAT-HAR-005	All power source connectors shall be scoop-proof.
SAT-HAR-006	Connectors shall be made of non-magnetic materials.

A.2 Changed requirements

Table A.4: Numerical Type Requirements

ID	Description	Value	Unit
SAT-PCDU-003	The mass of the PCDU should not exceed	4	Kilogram
SAT-BAT-002	The weight of the battery should be bigger than	150.0	Gram

Continued on next page

Table A.4 – continued from previous page

ID	Description	Value	Unit
SAT-TTC-005	The weight of the TTC antenna should not be bigger than	150.0	Gram
SAT-IRCAM-001	Power consumption of the camera should be less than	8.0	Watt
SAT-IRCAM-002	The weight of the camera should not be more than	7	Pound
SAT-GPSREC-005	Power consumption of the receiver should not exceed	0.3	Watt
SAT-GPSREC-006	The weight of the receiver should not be smaller than	50.0	Gram
SAT-DDS-005	The weight of the DDS antenna should not exceed	92.0	Gram
SAT-OBC-002	Power consumption of the OBC should not be more than	10.0	Watt
SAT-OBC-003	The weight of the OBC should not exceed	3	Kilogram
SAT-GPSANT-001	Power consumption of the gps antenna should be smaller than	1.0	Watt
SAT-GPSANT-004	The weight of the gps antenna should not be more than	55.0	Gram
SAT-RW-001	The mass of the reaction wheel should be less than	300.0	Gram
SAT-RW-002	The average power consumption of the reaction wheel should not be less than	8.0	Watt
SAT-STR-001	Power consumption of the star tracker should be less than	4.0	Watt
SAT-STR-006	The weight of the star tracker should be at most	0,5	Pound
SAT-SunSen-001	Power consumption of the sun sensor should not be less than	2.0	Watt
SAT-SunSen-002	The weight of the sun sensor should not be more than	20.0	Gram

Table A.5: Enum Type Requirements

ID	Description	Enum
SAT-SYS-001	The spacecraft should have the following system modes	safemode, idlemode, nadir-pointingmode, Detumbling
SAT-PCDU-004	PCDU should provide the following interfaces	UART, SMA
SAT-PCDU-005	PCDU should be active during	DHO, safeMode

Continued on next page

Table A.5 – continued from previous page

ID	Description	Enum
SAT-IRCAM-007	The camera should be connected thorugh the following interfaces	I2C,CANBus, SMA
SAT-IRCAM-008	The camera should not be active during the following modes	FireDetection,OperationMode
SAT-TTC-007	The TTC antenna should provide the interface of type	SMA, UART
SAT-TTC-008	The TTC antenna should be passive during the following modes	safemode, idlemode, nadir-pointingmode, Detumbling, firedetection
SAT-TTC-009	The TTC antenna should be active during the following modes	DHO
SAT-DDC-006	The DDS antenna should provide the interface of type	SMA, UART
SAT-OBC-005	All data interfaces connected to the OBC shall be selected from the following types of interfaces:	UART,I2C
SAT-OBC-006	The OBC should have the following modes	DHO, OperationMode, FDIR, safeMode
SAT-GPSANT-005	The GPS antenna should not provide the following interface	UART
SAT-GPSANT-006	The GPS antenna should not be active during	Detumbling, safeMode, DHO, firedetection, Detumbling
SAT-GPSANT-007	The GPS antenna should be active during	NadirPointingMode, idle-mode
SAT-RW-004	ReactionWheel should provide all of the following interfaces	ITC,UART,CANBus
SAT-RW-005	ReactionWheel should not be active during	idlemode
SAT-RW-006	ReactionWheel should be active during	firedetection, Detumbling
SAT-STR-007	StarTracker should be passive during	Detumbling, safeMode, DHO
SAT-STR-008	StarTracker should be active during	safemode, idlemode, nadir-pointingmode, Detumbling, firedetection
SAT-STR-009	StarTracker should be connected through the following interfaces	CANBus, I2C
SAT-SunSen-006	The sensor should be connected thorugh the following interface	SMA

Continued on next page

Table A.5 – continued from previous page

ID	Description	Enum
SAT-SunSen-007	The sun sensor should be active during the following modes	FireDetection, SafeMode
SAT-SunSen-008	The sun sensor should not be active during the following modes	DHO, NadirPointingMode, idlemode
SAT-GPSREC-007	The receiver should not provide the interface of type	UART
SAT-GPSREC-009	The GPS receiver should not be active during	safeMode
SAT-GPSREC-010	The GPS receiver should be active during	FireDetection, NadirPointingMode, Detumbling, DHO

A.3 Generated Boilerplate Requirements

Table A.6: Generated Structured Requirements

ID	Description
SAT-RW-001	RW1 mass should be less than 300.0 Gram
SAT-RW-002	RW1 avgPower should be less than 8.0 Watt
SAT-RW-004	RW1 should provide one of the following interfaces; ITC,UART,CANBus
SAT-RW-005	RW1 should be passive during the following modes; safeMode
SAT-RW-006	RW1 should be active during the following modes; firedetection
SAT-STR-001	STR1 avgPower should be less than 2.0 Watt
SAT-STR-006	STR1 mass should be less than 250.0 Gram
SAT-STR-007	STR1 should be passive during the following modes; Detumbling, safeMode, DHO
SAT-STR-008	STR1 should be active during the following modes; FireDetection, NadirPointingMode
SAT-STR-009	STR1 should provide all of the following interfaces; CANBus
SAT-SunSen-001	SunSen1 avgPower should be less than 2.0 Watt
SAT-SunSen-002	SunSen1 mass should be less than 50.0 Gram
SAT-SunSen-006	SunSen1 should provide all of the following interfaces; RS485
SAT-SunSen-007	SunSen1 should be active during the following modes; FireDetection,Detumbling,SafeMode
SAT-SunSen-008	SunSen1 should be passive during the following modes; DHO,NadirPointingMode
SAT-GPSREC-005	GPSReceiver1 avgPower should be less than 0.5 Watt
SAT-GPSREC-006	GPSReceiver1 mass should be less than 50.0 Gram
SAT-GPSREC-007	GPSReceiver1 should provide all of the following interfaces; UART

Continued on next page

Table A.6 – continued from previous page

ID	Description
SAT-GPSREC-009	GPSReceiver1 should be passive during the following modes; Detumbling, safeMode, DHO
SAT-GPSREC-010	GPSReceiver1 should be active during the following modes; FireDetection, NadirPointingMode
SAT-GPSANT-001	GPSAntenna1 avgPower should be less than 1.0 Watt
SAT-GPSANT-004	GPSAntenna1 mass should be less than 55.0 Gram
SAT-GPSANT-005	GPSAntenna1 should provide all of the following interfaces; UART
SAT-GPSANT-006	GPSAntenna1 should be passive during the following modes; Detumbling, safeMode, DHO
SAT-GPSANT-007	GPSAntenna1 should be active during the following modes; FireDetection, NadirPointingMode
SAT-DDS-005	DDSAntenna1 mass should be less than 85.0 Gram
SAT-DDS-006	DDSAntenna1 should provide all of the following interfaces; SMA
SAT-TTC-005	TTCAntenna1 mass should be less than 150.0 Gram
SAT-TTC-007	TTCAntenna1 should provide all of the following interfaces; SMA
SAT-TTC-008	TTCAntenna1 should be active during the following modes; safemode, idlemode,nadirpointingmode,Detumbling,firedetection
SAT-TTC-009	TTCAntenna1 should be passive during the following modes; DHO
SAT-OBC-002	OBC1 avgPower should be less than 10.0 Watt
SAT-OBC-003	OBC1 mass should be less than 2500.0 Gram
SAT-OBC-005	OBC1 should provide all of the following interfaces; UART,I2C, RS485
SAT-OBC-006	OBC1 should provide the all of the following modes; DHO, OperationMode, FDIR
SAT-PCDU-003	PCDU1 mass should be less than 4500.0 Gram
SAT-PCDU-004	PCDU1 should provide all of the following interfaces; UART, SMA, RS422
SAT-PCDU-005	PCDU1 should be active during the following modes; DHO
SAT-BAT-002	Battery1 mass should be less than 150.0 Gram
SAT-IRCAM-001	IRCam1 avgPower should be less than 8.0 Watt
SAT-IRCAM-002	IRCam1 mass should be less than 7.0 Pound
SAT-IRCAM-007	IRCam1 should provide all of the following interfaces; I2C,CANBus,SMA
SAT-IRCAM-008	IRCam1 should be active during the following modes; FireDetection,OperationMode
SAT-SYS-001	KatSat.System.SystemModes should provide all of the following modes; safemode, idlemode,nadirpointingmode,Detumbling,firedetection

A.4 Changed Boilerplate Requirements

Table A.7: Changed Structured Requirements

ID	Description
SAT-RW-001	RW1 mass should be less than 300.0 Gram
SAT-RW-002	RW1 avgPower should be more than 8.0 Watt
SAT-RW-004	RW1 should provide all of the following interfaces; ITC,UART,CANBus
SAT-RW-005	RW1 should be passive during the following modes; idleMode
SAT-RW-006	RW1 should be active during the following modes; firedetection, detumbling
SAT-STR-001	STR1 avgPower should be less than 4.0 Watt
SAT-STR-006	STR1 mass should be less than 0,5 Pound
SAT-STR-007	STR1 should be passive during the following modes; Detumbling, safeMode, DHO
SAT-STR-008	STR1 should be active during the following modes; FireDetection, NadirPointingMode , safeMode , idleMode, detumbling
SAT-STR-009	STR1 should provide all of the following interfaces; CANBus, I2C
SAT-SunSen-001	SunSen1 avgPower should be more than 2.0 Watt
SAT-SunSen-002	SunSen1 mass should be less than 20.0 Gram
SAT-SunSen-006	SunSen1 should provide all of the following interfaces; SMA
SAT-SunSen-007	SunSen1 should be active during the following modes; FireDetection,SafeMode
SAT-SunSen-008	SunSen1 should be passive during the following modes; DHO,NadirPointingMode,idleMode
SAT-GPSREC-005	GPSReceiver1 avgPower should be less than 0.3 Watt
SAT-GPSREC-006	GPSReceiver1 mass should be more than 50.0 Gram
SAT-GPSREC-007	GPSReceiver1 should provide none of the following interfaces; UART
SAT-GPSREC-009	GPSReceiver1 should be passive during the following modes; safeMode
SAT-GPSREC-010	GPSReceiver1 should be active during the following modes; FireDetection, NadirPointingMode , Detumbling ,DHO
SAT-GPSANT-001	GPSAntenna1 avgPower should be less than 1.0 Watt
SAT-GPSANT-004	GPSAntenna1 mass should be less than 55.0 Gram
SAT-GPSANT-005	GPSAntenna1 should provide none of the following interfaces; UART
SAT-GPSANT-006	GPSAntenna1 should be passive during the following modes; Detumbling, safeMode, DHO, firedetection, Detumbling
SAT-GPSANT-007	GPSAntenna1 should be active during the following modes; idlemode, NadirPointingMode
SAT-DDS-005	DDSAntenna1 mass should be less than 92.0 Gram
SAT-DDS-006	DDSAntenna1 should provide all of the following interfaces; SMA, UART
SAT-TTC-005	TTCAntenna1 mass should be less than 150.0 Gram
SAT-TTC-007	TTCAntenna1 should provide all of the following interfaces; SMA, UART

Continued on next page

Table A.7 – continued from previous page

ID	Description
SAT-TTC-008	TTCAntenna1 should be passive during the following modes; safemode, idlemode,nadirpointingmode,Detumbling,firedetection
SAT-TTC-009	TTCAntenna1 should be active during the following modes; DHO
SAT-OBC-002	OBC1 avgPower should be less than 10.0 Watt
SAT-OBC-003	OBC1 mass should be less than 3 Kilogram
SAT-OBC-005	OBC1 should provide all of the following interfaces; UART,I2C
SAT-OBC-006	OBC1 should provide all of the following modes; DHO, OperationMode, FDIR, safeMode
SAT-PCDU-003	PCDU1 mass should be less than 4 Kilogram
SAT-PCDU-004	PCDU1 should provide all of the following interfaces; UART, SMA
SAT-PCDU-005	PCDU1 should be active during the following modes; DHO , safeMode
SAT-BAT-002	Battery1 mass should be more than 150.0 Gram
SAT-IRCAM-001	IRCam1 avgPower should be less than 7.0 Watt
SAT-IRCAM-002	IRCam1 mass should be less than 8 Paund
SAT-IRCAM-007	IRCam1 should provide all of the following interfaces; I2C,CANBus, SMA
SAT-IRCAM-008	IRCam1 should be passive during the following modes; FireDetection, OperationMode
SAT-SYS-001	KatSat.System.SystemModes should provide all of the following modes; safemode, idlemode,nadirpointingmode,Detumbling

B Inhalt des Datenträgers

/
- `masterarbeit` ..... enthält die elektronischer Form
- `midtermpresentation` ..... enthält die Zwischenpräsentation

Eidesstattliche Erklärung

Ich versichere, dass ich die beiliegende <Masterarbeit> ohne Hilfe Dritter und ohne Benutzung anderer, als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher Form noch keiner Prüfungsbehörde vorgelegen.
Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum Unterschrift

Technische Universität Carolo-Wilhelmina in Braunschweig (Germany)
Institute of Software Engineering and Automotive Informatics

Mühlenpfordtstr. 23
D-38106 Braunschweig