



Multi-Satellite Mission
Operations System
Providing Constellation
Planning Functionality



Kai Leidig

Doctoral Thesis

**Multi-Satellite Mission
Operations System
Providing
Constellation Planning Functionality**

A thesis accepted by
the Faculty of Aerospace Engineering and Geodesy of
the University of Stuttgart
in partial fulfillment of the requirements for the degree of
Doctor of Engineering Sciences (Dr.-Ing.)

by

Kai Leidig

born in Engelskirchen, Germany

Main referee: *Prof. Dr.-Ing. Jens Eickhoff*

Co-referee: *Prof. Dr.-Ing. Reinhold Bertrand*

Co-referee: *Prof. Dr.-Ing. Sabine Klinkner*

Date of Defense: *January 9, 2023*

Institute of Space Systems
UNIVERSITY OF STUTTGART
2023

Abstract

According to established space engineering standards, a space system can be decomposed into two main parts: the space segment, and the ground segment. While most space segments used to consist of just one single satellite, this thesis considers the growing case of a space segment featuring an arbitrary multiple of satellites. This is the case under two circumstances: The space segment is either a constellation, or a number of independent satellites are operated in parallel. A constellation describes a cluster of satellites all contributing to a common objective, while the case of one organization operating multiple independent satellites simultaneously is referred to as multi-mission operations.

Regardless of which of the two cases applies, simultaneous operation of multiple satellites leads to a series of problems. The more satellites are operated, the more compete these satellites for the limited resources required for executing the mission, such as ground station time or operations personnel. A further problem is the one caused by the increased complexity. This work refers to the term complexity as a measure for system functionality and the amount of interactions between different systems.

Existing satellite operations concepts are usually data driven, which means that the entire design primarily focusses on the creation of a data connection between space and ground. Commu-

nication with the satellite is based on the exchange of telemetry and telecommands, enabling ground to access certain services in the satellite on-board software. Software establishing such a connection must then be complemented with a series of systems, which are furthermore required for satellite operations, including but not limited to flight dynamics, mission planning, or automation approaches.

With respect to multi-satellite operations, this causes a series of difficulties. Satellite operations software is usually customized for each mission. Nonetheless, these products are commonly derived from former solutions for single-satellite operations and do not properly consider those problems that arise when there are multiple satellites to be operated.

In other words, common implementations always fall back on the same old solutions, which are then adapted for the respective mission. These old solutions are not designed for constellation or multi-mission operations though, which is why they do not properly reflect the requirements of such a use case. As a consequence, these systems become prone to error. The latter is caused by not so-phonically dealing with the phenomenon of an increased complexity, respectively by not adequately mapping processes and the states of the operated systems.

This thesis addresses the design of a system primarily designed for the operations of multiple satellites simultaneously. The work is done under the principle hypothesis that a complex system cannot be operated efficiently without automation. Unlike existing solutions, this work does not refer to automation as an additional ground system feature, but as a holistic process that involves the entire system including the operated satellites. Consequently, the proposed concept is not data driven anymore, but process driven.

Automation as a form of controlling a system requires knowledge on the state of the operated system. Due to the natural limitations of satellite communications, that information is not continuously available. In order to compensate for data gaps, this work proposes a concept by means of which the satellite system state can be mapped, planed, estimated, and verified on ground. The same concept further implements an approach for the planning of activities across system boundaries, which allows for the coordination of tasks between multiple collaborating systems within a network, and thus for the planning of constellation and multi-satellite activities too.

Kurzfassung

Gemäß einschlägiger Standards setzt sich ein Raumfahrtsystem aus zwei Komponenten zusammen: dem Raumsegment und dem Bodensegment. Bisher bestanden die meisten Raumsegmente aus lediglich einem einzelnen Satelliten. Die vorliegende Arbeit widmet sich dem jedoch immer häufiger auftretenden Fall, dass sich das Raumsegment aus einer beliebigen Vielzahl an Satellitensystemen zusammensetzt. In zwei Situationen ist dies der Fall. Erstens, beim Raumsegment handelt es sich um eine Konstellation, also um einen Verbund mehrerer an einer gemeinsamen Aufgabe beteiligter Systeme, oder zweitens, es besteht Multi-Missionsbetrieb. Im zweiten Fall werden mehrere voneinander unabhängige Satellitensysteme gleichzeitig von einer einzelnen Organisation betrieben.

Durch den Betrieb mehrerer Satelliten gleichzeitig, ergibt sich das Problem der Konkurrenz um eine begrenzte Menge an für die Durchführung der Mission(en) zur Verfügung stehender Ressourcen, wie etwa Bodenstationen oder Personal für den Betrieb. Ein weiteres Problem ist jenes der gestiegenen Komplexität des betriebenen Systems. In dieser Arbeit wird der Begriff Komplexität als ein Maß für die Menge an Systemfunktionalität und die Interaktion zwischen verschiedenen Systemen heran gezogen.

Bestehende Konzepte für den Satellitenbetrieb sind in der Regel datenbasiert. Das heißt, es wird sich konzentriert auf die Herstellung

einer Kommunikation mit dem Satelliten auf Grundlage von Telemetrie und einzelner Telekommandos, mit Hilfe derer sich etwa Dienste in der Satelliten on-board Software steuern lassen. Für den Betrieb muss diese zentrale Software dann um eine ganze Reihe an Systemen ergänzt werden. Dazu zählen unter anderem Flugdynamik, Missionsplanung, oder Ansätze zur Automatisierung.

Für den Multi-Missionsbetrieb ergeben sich dadurch einige Schwierigkeiten. Bei Software im Bodensegment, insbesondere im Betrieb, handelt es sich in der Regel um Sonderanfertigungen. Jedoch setzen diese meist auf Systemen auf, welche einst für den Einzelbetrieb konzipiert wurden und die nicht die Probleme berücksichtigen, welche erst durch eine Mehrzahl an zu betreibenden Satelliten entstehen.

Mit anderen Worten: Bei der Implementierung von Systemen für den Satellitenbetrieb wird auf die immer selben Softwarelösungen zurückgegriffen und diese dann an die jeweilige Mission angepasst. Diese sind jedoch nicht für den Mehrsatellitenbetrieb ausgelegt, wodurch sie den damit einhergehenden Anforderungen nicht ohne weiteres gerecht werden können, was diese Systeme entsprechend fehleranfällig macht. Letzteres wird zum Beispiel dadurch hervorgerufen, dass dem Phänomen der gestiegenen Komplexität nicht angemessen Rechnung getragen wird und Vorgänge, die mehrere interagierende Systeme mit einbeziehen nicht adäquat abgebildet werden können.

Ziel der vorliegenden Arbeit ist der Entwurf eines Systems, welches von Anfang an für den simultanen Betrieb mehrerer Satellitensysteme ausgelegt ist. Dabei wurde die Grundannahme getroffen, dass Systeme ab einem gewissen Grad an Komplexität effizient nur noch automatisiert betrieben werden können. Im Gegensatz zu einschlägigen Lösungen wird Automatisierung jedoch nicht als eine zusätzliche Funktionalität des Betriebssystems angesehen, sondern als Prozess, welcher das gesamte Betriebssystem einschließlich

der betriebenen Satelliten umfasst. Folglich ist das ausgearbeitete Konzept nicht mehr rein daten- sondern vorrangig prozessorientiert.

Automatisierung also Form der Regelung eines Systems erfordert außerdem die Verfügbarkeit des Systemzustands. Naturgemäß liegt diese Information bei einem Satelliten nicht kontinuierlich vor. Aus diesem Grund wird neben der globalen Betriebssystemarchitektur ein Konzept entworfen mithilfe dessen sich der Zustand des betriebenen Satelliten auf Systemebene planen, vorhersagen und verifizieren lassen kann. Um darüber hinaus sowohl der Aufgabe des Konstellations- als auch des Multi-Missionsbetriebs gerecht zu werden, ermöglicht das ausgearbeitete Konzept zudem das Planen von Vorgängen über Systemgrenzen hinweg und damit die Koordinierung der Zusammenarbeit mehrerer Systeme in einem Verbund.

Acronyms & Abbreviations

ACK	acknowledgment
AIS	Automatic Identification System
Ana.	Analysis
AOI	area of interest
AOS	acquisition of signal
API	Application Programming Interface
APID	Application Process ID
App	Application
AS	Automation System
BCBF	body-centered body-fixed
BCI	body-centered inertial
CCS	Central Checkout System
CCSDS	Consultative Committee for Space Data Systems
CLTU	Command Link Transmission Unit
CMD	command
Com.	communication
COTS	commercial off-the-shelf
CPT	Constellation Planning Tool
CPU	Central Processing Unit

DAS	Data and Archives System
DLR	German Aerospace Center
E2E	End-to-End
ECEF	earth-centered earth-fixed
ECI	earth-centered inertial
ECSS	European Cooperation for Space Standardization
EnvDyn	Environment and Dynamics Models
EO	Earth Observation
FDIR	Failure Detection Isolation and Recovery
FDS	Flight Dynamics System
FDT	Flight Dynamics Tool
GFZ	German Research Center for Geosciences
GNC	Guidance Navigation and Control
GPS	Global Positioning System
GS	Ground Station
GSOC	German Space Operations Center
GUI	Graphical User Interface
I/O	input / output
ICD	Interface Control Document
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IF	interface
Insp.	inspection
IOT	Internet of Things

IRS	University of Stuttgart, Institute of Space Systems
ISL	inter-satellite link
ITU	International Telecommunication Union
JSON	JavaScript Object Notation
LEO	low Earth orbit
LEOP	Launch and Early Orbit Phase
LLA	latitude, longitude and altitude
LOAT	Level of Automation Taxonomy
LOS	loss of signal
LTAN	local time of ascending node
MMOS	Multi-Mission Operations System
MCS	Mission Control System
MCT	Mission Control Tool
MGSE	Mechanical Ground Support Equipment
MIB	Mission Information Base
MJD	Modified Julian Date
MMU	Mass Memory Unit
MOCR	Mission Operations Control Room
MOIS	Manufacturing and Operations Information System
MOM	Message Oriented Middleware
MPS	Mission Planning System
MPT	Mission Planning Tool
Nav.	navigation
NCTRS	ESA Network Controller and Telemetry Router System
NIS	ESA Network Interface System

OBC	On-board Computer
OS	Operating System
OSI	Open Systems Interconnection
Prio.	priority
PUS	Packet Utilization Standard
RAAN	right ascension of the ascending node
Rev.	review
RMAP	Remote Memory Access Protocol
RTEMS	Real-Time Executive for Multiprocessor Systems
RTG	Radioisotope Thermoelectric Generator
RX	receiver
S/C	spacecraft
Sci.	science
SCID	Spacecraft ID
SCM	Source-Control Management
SGP4	Simplified General Perturbation Model
SIB	System Information Base
SimTG	Simulation Third Generation
SLE	ESA Space Linke Extension
SMP2	Simulation Model Portability Standard (version 2)
SMT	Simulated Mission Time
SOA	Service-Oriented Architecture
SPARC	Scalable Processor Architecture
SPL	Software Product Line
SRDB	Satellite Reference Database

SRT	Simulation Runtime
SSOT	single source of truth
SST	Simulator Session Time
TC	Telecommand
TCP/IP	Transmission Control Protocol / Internet Protocol
TLE	Two Line Element
TM	Telemetry
TX	transmitter
UML	Unified Modeling Language
UPM	Universiti Putra Malaysia
UTC	Universal Time Code
UUID	Universally Unique Identifier
Ver.	verification
VGS	Virtual Ground Station
VM	Virtual Machine
XTCE	XML Telemetric and Command Exchange

Contents

Abstract	iii
Kurzfassung	vii
Acronyms & Abbreviations	xi
1 Introduction	1
1.1 Motivation	2
1.2 Best Practices in Constellation Operations	4
1.3 Scope and Research Hypothesis	7
1.4 Structure – A Guide Through this Thesis	8
2 Background	11
2.1 Flying Laptop	12
2.1.1 Space Segment	12
2.1.2 Ground Segment	14
2.2 Upcoming Missions	17
2.2.1 EIVE	17
2.2.2 SOURCE	18
2.2.3 Ground Station Network	19
2.3 The New Operations System	20
3 Objectives	23
3.1 Paradigm Shifts in Satellite Mission Operations	23
3.2 Reference Mission Architecture	25
3.2.1 Size	28
3.2.2 Spacial Distribution	31

3.2.3	Operating Principle	32
3.2.4	Architectural Homogeneity	32
3.2.5	Service Availability	34
3.3	Summary	35
3.3.1	Top-Level Requirements	36
4	Methodology	39
4.1	The Wife and the Mother-in-Law	39
4.2	Project Organization	41
4.2.1	Agile vs. Process Oriented	42
4.2.2	Agile Methods	43
4.3	Domain Engineering	48
4.3.1	Domain Analysis	52
4.3.2	Domain Design	56
4.4	Implementation	57
4.4.1	Selection of a Programming Language	58
4.4.2	Source-Control Management	60
4.5	Test and Verification	62
4.5.1	Terminology and Fundamental Aspects of Software Testing	63
4.5.2	Test Items and Verification Process	69
4.5.3	System Simulation	76
5	MMOS Domain Engineering	89
5.1	Quality Requirements	89
5.1.1	Runtime Requirements	91
5.1.2	Non-Runtime Requirements	108
5.2	Domain Analysis	112
5.2.1	Space System Breakdown	112
5.2.2	MMOS Subsystems	115
5.3	Design	142
5.3.1	System Architecture	142
5.3.2	Interfaces	153
5.3.3	System Configuration	170

6	Mission Planning	179
6.1	Value of the Mission Planning Tool	179
6.2	The Activity	181
6.2.1	Inheritance	184
6.2.2	Class Diagram	186
6.2.3	Resource Demand	188
6.2.4	State	198
6.2.5	Nested Activities	206
6.2.6	Derived Activities (Examples)	206
6.3	Mission Planning Tool Architecture	212
6.3.1	Configuration Layer	212
6.3.2	Scheduling Layer	214
6.3.3	Operative Layer	217
6.3.4	Interface Layer	221
6.3.5	Communication Layer	222
6.4	The Schedule	223
6.4.1	Graphical User Interface	225
6.5	Constellation Planning	227
6.5.1	Layout	228
6.5.2	Reflection of Architecture Requirements	233
7	System Test and Verification Environment	237
7.1	System Simulation	238
7.1.1	Model Libraries	238
7.1.2	OBC Emulation	245
7.2	Simulation Infrastructure	256
7.2.1	Simulator Checkout System	257
7.2.2	System Simulation Network	258
7.3	Simulated Mission	260
7.3.1	Scenario	262
7.3.2	Final Setup	266
8	Summary	269

9 Outlook	273
A Information Used	277
A.1 List of Satellite Constellations	277
A.2 The LOAT Matrix	283
B MPT Implementation	291
B.1 Mission Planning Tool (MPT) Requirement Backlog	291
B.2 MPT Activity Management Flow Diagrams	309
C Space Debris	323
Bibliography	325
Index	343

Introduction

After more than sixty years of spacecraft operations, aerospace community still knows quite little about satellite constellation operations. This is due to several reasons. Constellations are complex, expensive and used to be operated by agencies, large companies or syndicates only. The term *constellation* is actually not well defined, even though it is used by media, and an increasing number of new space companies has launched satellite constellations within the last decade [11].

Today, several approaches of defining a satellite constellation exist. The most simple approach is defining a constellation by the pure number of satellites. Thus, two satellites can be a satellite constellation already. However, the general idea of a constellation is that multiple satellites “work together to achieve common objectives” ([134], p. 269). This means that not only the number of satellites make a constellation, but also the fact that those satellites are dedicated to a common objective.

Beyond that, constellations are further distinguished by the number and the shape of the orbits. Satellites within about the same orbit and performing relative station-keeping are commonly referred to as a *formation*, where the term constellation describes a set of satellites performing independently on separate orbits ([134], p. 269).

Other definitions go further and claim that a major aspect of a constellation is also the functional similarity of the satellites ([53], p. 26).

In accordance with [134] this thesis defines a constellation as follows:

A constellation is a system that consists of multiple satellites, which provide functionality to serve a common objective.

This definition completely disregards the number and the shape of the orbits, the functional implementation of the satellites or any kind of satellite interaction. A more comprehensive definition of the term and a discussion of other special forms will be pursued in the later course of this work.

This work intends to regard a satellite constellation in a very macroscopic manner and from a systematic point of view, because the overall goal is the proposal of operations system architecture and not the design of a constellation. Moreover, this thesis intends to handle aspects of constellation operations and fleet management in the most possible generic fashion. Nevertheless, the way how design, size, or distribution of a constellation affect the design of such a system must be discussed.

1.1 Motivation

Operating a constellation raises some problems. First of all, constellations are complex systems. But what does that mean? The term complexity is used intensively on many occasions, usually when things tend to get difficult to handle. Definitions vary depending on the nature of the described system and the field of science. Hence, different definitions of complexity are considered in the fields of nature science, computer science, social science etc.

The Duden dictionary defines a complex as “a structured unit, which can be decomposed into components and domains”¹ [132]. Accordingly, one author defines complexity as a means of characterizing “the behavior of a system [...] whose components interact in multiple ways and follow local rules ...” [76, 136]. In other words,

a system as a functional unit is more complex the more components it consists of and the more these components interact with each other.

Since satellites can also be quite difficult to handle, it makes sense to structure space missions into separate functional domains (systems) and therefore to provide means of dealing with the complexity of a mission. The European Cooperation for Space Standardization (ECSS) decomposes a space mission into three main areas. The *Space Segment*, the *Ground Segment*, and the *Launch Segment*. Each of these areas can be subdivided further into systems, subsystems, components and elements [37]. This thesis focuses on the ground segment and the space segment, where the launch segment is out of scope of this elaboration.

Compared to a single satellite mission, the higher level of complexity in a constellation mission is obvious. In a single satellite mission the space segment consists of one system, the satellite. In a constellation the space segment consists of n satellites. If complexity could be quantified, it would be n times higher, presumably. That rise would be even larger in case these satellites are also functionally coupled in some manner.

Resources in an operations center are limited, which is why an operations system suited for constellation operations must somehow incorporate that increased complexity. An operations system that is well designed for single satellite missions does not necessarily be

¹German definition translated into English

capable of operating a constellation as well, because these systems are usually too slow and inefficient.

For instance, during Launch and Early Orbit Phase (LEOP), spacecraft operations is less automated and binds a lot of personnel ([128], sec. 2.1.3). During that phase the space segment is checked out and prepared for its upcoming mission. At any time, but especially during LEOP, an operator must focus on its pure task and must not be impeded by an unnecessary workload. If a problem or a cause for an action is identified, that action must be implemented and executed efficiently, without losing valuable time. On the other hand, an operator or an automatic decision-making must be provided with the required data products. In the event of a system failure, time consuming data analyses that still work to recover one satellite could be hazardous for a constellation.

To sum up, the major problem of constellation operations is that constellations are nested and interactive systems and therefore very complex. This level of complexity must be covered by the operations system. In order to deal with that situation, an operations system must be very efficient in the process of handling the operated spacecrafts. This requires the implementation of a lean control process, including a proper mapping of the satellites' states on ground.

1.2 Best Practices in Constellation Operations

If efficiency is measured by the size of staff that is required to operate the mission, the most obvious solution for increasing efficiency is automation. In their paper about *Best Practices for Operations of Satellite Constellations* Joseph Howard et al. mention automation as one of the key means of operating a constellation [67]. The paper lists a number of best practices, that were identified based on discussions the authors did with constellation operators in the year 2005. The

authors distinguish between the autonomy of the spacecraft and the automation to be implemented on ground.

In accordance with the general usage of the terms, this elaboration defines *autonomy* and *automation* as follows:

- Autonomy describes the capability of one system (the satellite) to operate itself without the need of external monitoring and command. Autonomy is a *quality* of a system, while
- automation describes a *process*. Automation enables one system (the operations system) to operate and to control another system (the satellite) without the need of human interaction.

Furthermore, this work distinguishes between the terms *operations* and *control* as follows:

- Operations covers all processes on ground which are needed for the achievement of the mission goals, including the monitoring, the resolution, and the control of the state of the operated system.
- Controlling describes a closed technical process of actively managing the state of a *controlled* system. More precisely, it refers to a feedback process, where a system state is maintained based on existing knowledge about that state.

Howard et al. recommend that “the best practice is to put automation on board the spacecraft” ([67], p. 8) and thus speak in favor of spacecraft autonomy rather than ground system automation. Yet, with the limited resources available on board a satellite, such as sensor measurements, computing performance etc., this is only feasible for quite simple tasks. Therefore, even [67] states that for some activities, an automatic ground segment is necessary. This last statement is according to the design principles implemented by operators of large swarms. According to [103], constellation autonomy can also be achieved by implementing high-level automation in

all necessary areas of the ground segment.

The problem with automation is though that it is something that cannot be added to a system like some sort of additional feature. As mentioned above, automation refers to a process, not to a subsystem, unit, or element.

Almost every mission operations solution commercially available claims to support automation. Most of them are built based on one or the other established Mission Control System (MCS)². The principle concept of these MCS dates back into a time were spacecraft were operated with minimum support by a computer. Back then operations was done by human operators sending single TCs to the satellite, and evaluating the received TM on simple displays. Over time, more and more features were added to these MCS, features like on-board service management, event management, improved telemetry dashboards, and eventually *automation*.

What is referred to as automation though does not describe a fully automatic spacecraft control process, but single software features fulfilling tasks formerly done by a human: tasks like generating a command stack from a schedule, reacting on specific events, creating reports and notifications, execution of specific procedures, etc.

Fully automating operations on the basis of such systems requires a lot of effort and adaption of the concept. What usually comes out of that is some sort of of automation, which [99] refers to as “clumsy”. A characteristic of such systems is that they are difficult to engage and prone to error.

What works for single satellite operations does not necessarily work for constellations as well and definitely not for multi-mission operations, as most of these systems completely fall apart, or need to be reworked intensively when a new mission needs to be operated.

²The term MCS has evolved historically and is quite misleading in this particular context, because what these systems provide is actually no control functionality as defined above. In modern operations concepts MCS refers to the system that exchanges TM/TC with the operated system.

1.3 Scope and Research Hypothesis

Scope of this thesis is the proposal of an architecture for a multi-mission operations system, including an appropriate mission planning concept. Both, global architecture and mission planning concept shall be designed for constellation operations.

The work is done under the general assumption that automation is only possible if the operated system can be monitored and controlled by a closed technical system. Monitoring and control algorithms in turn depend on the availability of valid data about that system's state.

The problem with monitoring a satellite is telemetry not being continuously available due to the natural, physical separation of space and ground segment. Any sort of automation must compensate for the lacking of current system data. Thus, a mission planning concept must provide means of creating a forecast of the system state based on which a new planning can be made. In turn, scheduled activities must be considered by the forecasting, as they actively alter the state of the operated system. Thus, the first research hypothesis is the following:

If the system state can be modeled, resolved, and tracked sufficiently precise by a closed system, automatic satellite operations can be possible without the presence of real-time monitoring data.

As stated in the very beginning of this chapter, a constellation is a system of satellites, which serve a common objective. This is why the operations system must be designed in a way that it supports that overall objective too. It is not sufficient if the operations system is just designed for the operations of a number of individual satellites. It must be capable of operating the constellation as *one* system. If the objective of a constellation is the provision of a service to a certain location on earth, operators surely do not want

to command each satellite individually, they want to command the entire constellation to provide that service and expect the operations system to do the rest of the commanding for them. The operations concept must therefore be capable of commanding the entire constellation, as well as each satellite individually, as it must be capable of dividing a global constellation schedule into separate satellite activities, which leads to a second research hypothesis:

If an individual satellite system can be operated automatically by a closed system, then it is also possible to operate a finite number of individual systems in order to serve a common, higher-level objective.

This thesis aims at the description of a software unit for the scheduling of constellation and constellation satellite activities. That unit shall be the Constellation Planning Tool (CPT). It shall fit into an operations system architecture with the demand for automation as specified above. In that context, the CPT shall prevent the satellite systems from getting into undefined states, as it shall prevent operators from scheduling conflicting activities. The CPT shall not provide any form of intelligence or active decision-making though. This shall be the task of higher-level planning agents, which are not discussed in detail during the course this work.

1.4 Structure – A Guide Through this Thesis

Chapters 2 & 3

The subsequent discussion begins with a brief overview on the background and the origin of the project. The chapter introduces the research facility where the work has been done, and justifies the development through the introduction of those satellite missions, which demand for an operations system.

The top-level requirements and the objectives derived from that demand are introduced in chapter 3. These first two chapters address those who are interested in why this project has been kicked-off, and what is expected from the envisaged operations system.

Chapter 4

A selection of applied methodologies is introduced in chapter 4. It focusses on those issues which were considered most relevant during the course of such a development project. The chapter deals with applied technical methodologies from the areas of software and domain engineering, but also non-technical issues like the applied project organization methodology. Addressed are readers particularly interested in how to approach a software development project in the domain of space mission operations. Readers, who are just interested in the details of the technical implementation can skip this chapter.

Chapter 5

In the course of this work, the methodology of domain engineering has been applied for the requirements specification as well as for the design of the Multi-Mission Operations System (MMOS). The outcome of this work is introduced in chapter 5. That chapter comprehensively covers the scope and the details of the system design. It begins with a discussion of the most relevant quality requirements, and elaborates in detail how these are reflected by the global architecture. Chapter 5 also deals with a comprehensive, but macroscopic discussion of the MMOS subsystems and their interfaces, as well as it tackles matters like system configuration and orchestration. It addresses anybody interested in what makes the developed system different from all the other mission operations systems, and how multi-mission capability can be achieved.

Chapter 6

The core of the proposed operations architecture is the mission planning approach. Therefore, chapter 6 goes into detail of the implementation of that subsystem. It introduces in particular the developed Activity concept for the realization of a reliable, and conflict-free schedule. It further introduces how schedules of multiple interacting systems can be managed, and how this approach supports constellation planning eventually. Compared to the previous chapter, chapter 6 goes more into low-level aspects of the implementation, and addresses everybody with a particular interest in how-to schedule various interacting systems of a space mission all together.

Chapter 7

This chapter discusses aspects of the system verification. A system like the introduced MMOS must be tested eventually, which is why a simulation infrastructure for the simulation of satellite constellations has been set up. Chapter 7 goes into details of the implementation of that simulation infrastructure, focussing on how it actively supports the verification of a multi-mission system. It particularly addresses readers with a special interest in satellite system simulation and on-board computer emulation for ground system verification. The chapter is less relevant for readers focussing on aspects of mission operations.

Chapters 8 & 9

This work closes with a summary of the presented results, and with an outlook for further development by highlighting the open issues and the potential of the developed architecture.

Background

Goal of this work is the conceptual development of a software tool for the mission planning of satellite constellations. That tool shall be embedded into a software architecture designed for the operations of multiple satellite missions in parallel, called the Multi-Mission Operations System (MMOS).

Before starting the development of an all new architecture it is worth making a step back though and have a look around what is already there and what is needed. This work was done at University of Stuttgart, Institute of Space Systems (IRS). Over the past years the institute gained a lot of experience in satellite system design, as well as the development of ground system architectures and systems for satellite operations. This experience has been the foundation based on which this work could be done.

Starting point for the development has been the *Flying Laptop* mission, respectively the operations system designed for that mission. *Flying Laptop* also was the starting point for subsequent satellite missions at the institute, which actually drove the development for a system like the MMOS, and which were the reason why this project was engaged eventually. This chapter justifies the need for the new operations concept, through a brief introduction of these missions.

However, the top-level objective of the envisaged system is multi-mission capability. Such a system must neither be designed against one mission, nor a particular satellite system architecture. Therefore, section 3 covers the definition of a general reference architecture against which the system can be designed instead.

2.1 Flying Laptop

On June 14, 2017 the small satellite *Flying Laptop* was launched from the Russian spaceport Baykonur. Since then, *Flying Laptop* has been operated from the premises of University of Stuttgart (IRS) [81]. *Flying Laptop* is the first satellite of the university's small satellites programme and the starting point of several subsequent missions initiated at the institute.

The *Flying Laptop* system consists of a space segment, a small satellite with a launched mass of approximately 110 kg in a sun-synchronous orbit ¹, and a ground segment featuring a fully operative control room and a ground station, both located at University of Stuttgart, Germany. The ground segment is extended by a series of globally distributed antennas, which belong to ground station networks of the German Aerospace Center (DLR) and the German Research Center for Geosciences (GFZ) [81].

2.1.1 Space Segment

Flying Laptop (fig. 2.1) was developed by the academic staff at the university. It marks also the initial design of an industrialized platform for low Earth orbit (LEO) satellites currently developed by Airbus.

The system features a series of payloads and experiments for earth observation and technology demonstration purposes, such as

¹LTAN \approx 09:30 h (UTC), Altitude \approx 600 km (Jan. 2021)

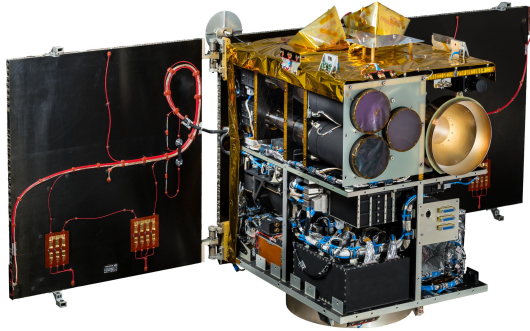


Figure 2.1: *Flying Laptop* - The image shows the fully integrated satellite, without its outer thermal insulation.

- diverse camera systems in the optical and the near infrared range,
- an Automatic Identification System (AIS) for the tracking of seagoing vessels,
- a GPS based attitude determination experiment,
- and a prototype of an optical data downlink system.

The latter was actually one of the design drivers for the satellite's attitude control system. Due to the narrow laser beam width, a pointing with an accuracy of less than 150 arc seconds is required for a successful link with an appropriate ground station [133].

A dedicated data downlink system is used for the transmission of the payload data. That data is transmitted in the amateur S-band, at a maximum rate of 10 *Mbit/s*. A further system is implemented for telemetry downlink and command uplink in the commercial S-band. That system provides a data rate of 128 *kbit/s* in the downlink and 4 *kbit/s* in the uplink [81, 15].

Core of the satellite bus is an On-board Computer (OBC), composed of two cold redundant LEON3FT processor boards, two cold



(a) Meteor Antenna



(b) IRS Control Room

Figure 2.2: Elements of the *Flying Laptop* Ground Segment, Located at the Institute of Space Systems, University of Stuttgart

redundant boards for TM storage and device I/O, and two hot redundant CCSDS-boards for TM data encoding and TC decoding [48]. The OBC processor board hosts the in-house built satellite on-board software, which controls the system. That on-board software is also the first implementation of a software framework that can be and has been adapted for further missions [9].

For the internal data handling tasks, the on-board software implements a series of services, which can be addressed from ground by means of the Packet Utilization Standard (PUS) protocol. This allows for live operations during the Stuttgart ground stations passes, as well as for the scheduling of tasks that happen offline. For that purpose, the on-board software features a queue that can be filled with up to approximately 1000 time-tagged commands.

2.1.2 Ground Segment

The *Flying Laptop* ground segment [81] features a ground station and a control room, both located at the institute's premises (fig. 2.2).

The 2.5 m parabolic antenna *Meteor* (fig. 2.2a) is installed on the roof of the control room building. Originally developed for a former application by the military, it has been redesigned to sup-

port satellite communication in the S-band. Therefore, a new feed had been integrated. That feed splits incoming right- and left-hand polarized signals, which allows for the reception of satellite TM and payload data simultaneously [15].

The meteor antenna facilitates *Flying Laptop* operations entirely from the university premises, although the ground station network features further antennas, distributed all over the globe. During Launch and Early Orbit Phase (LEOP), DLR antennas in Germany, northern Canada, and the Antarctic were used to check-out the space segment. These ground stations can still be used as fall-back systems in case of an emergency. A further GFZ ground station in Spitsbergen, Norway is used to extend TM and payload downlink capacity of the mission.

Via the university operations network, these ground stations are connected to the control room (fig. 2.2b). This room features all elements of a state-of-the-art satellite control facility. As such it provides workstations for the various tasks in the operations process:

- Flight Direction
- Subsystem Control (e.g. Power, GNC, Data, etc.)
- Payload Operations
- Command & Control
- Mission Planning
- Ground Station / Antenna Control
- Ground Systems Operations

Besides those hardware elements, the *Flying Laptop* ground segment of course features a series of software solutions too. These elements partly rely on commercial and/or third party products, as well as on tools that were developed in-house:

- An in-house built routing system supporting ESA Space Linke Extension (SLE), to connect to the *Meteor* ground station and the university ground station network
- *SCOS-2000* as Mission Control System (MCS)
- An in-house built database system for TM archiving and storage of payload data
- A ticketing system for mission planning, based on an open project planning software
- An in-house built flight dynamics tool for pass generation ²
- An in-house built multi-purpose display system for TM evaluation, health check, pass planning, etc.
- Several tool chains for tasks in the planning and scheduling process

Lots of these software components were designed according to the requirements of the *Flying Laptop* mission: for instance, the *Meteor* control software. Significant parts of the Mission Planning and the Display System were also tailored to the mission. This was mostly due to the fact that some needs became evident during operations, and a mission specific solution was normally the fastest in that situation.

After LEOP, when the mission entered its nominal operations phase, more and more tasks of the operations process were automated. It begun with the automatic generation of command stacks to be uplinked, and ended with the sophisticated scheduling of imaging activities, taking into account environmental conditions such as cloudiness, illumination conditions, etc.

All of these automatic features rely on automation scripts that needed to be implemented successively during the ongoing mission.

²The tool relies on third party Two Line Element (TLE) generation though.

As a result, the development was driven by the urgent demand for a working solution, rather than by the requirement for reusability. Accordingly, an adaption for another mission turned out to be extremely complicated.

2.2 Upcoming Missions

Based on the development, the lessons learned, and the resources from the *Flying Laptop* mission, University of Stuttgart is heading for a number of follow-on projects. Among of them are a couple of satellite missions, and a series of new antennas extending the university's ground station network.

2.2.1 EIVE

The satellite mission on *Exploratory In-orbit Verification of an E/W-band Link* (EIVE) is a collaborative project between Institute of Space Systems (IRS) and a consortium of other partners in industry and academia. *EIVE* is a six unit CubeSat featuring a 4K camera for high resolution images and a novel communication system to be tested in orbit.

Background of the mission has been the decision by the International Telecommunication Union (ITU) to release frequencies in the E/W band (71–76 GHz and 81–86 GHz) for satellite services, “which allows for data rates of several gigabits per second”, and further opens space industry for broadband communication and Internet of Things (IOT) applications [114]. Purpose of the mission is to demonstrate the capabilities of an appropriate satellite transmitter in orbit.

The in-orbit verification process is supposed to take place in three stages. At first, an arbitrary waveform is transmitted in order to study the basic functionality of the link. After that, real images of the 4K camera are going to be downlinked, in order to demonstrate

the pursued bandwidth. In a third step, the link is also verified for the transmission of satellite telemetry using state-of-the-art CCSDS compliant protocols.

Due to the nature of the mission, *EIVE* has also a huge impact on the university ground station. The mission is only feasible if the ground station provides E/W-band reception capabilities. Consequently, the university ground station has been extended by an appropriate antenna for the purpose (sec. 2.2.3).

The mission shall be operated for at least one year in a low earth orbit.

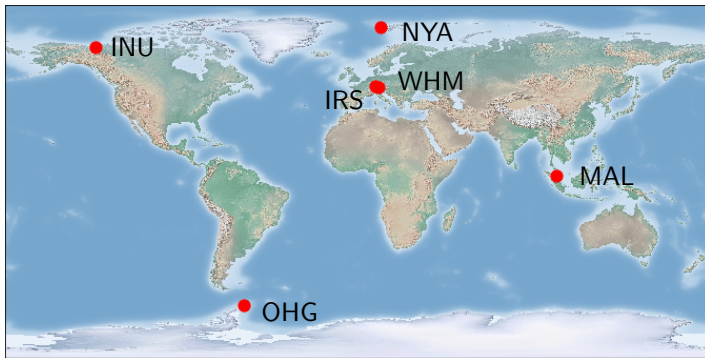
2.2.2 SOURCE

SOURCE is a three unit CubeSat mission by students from University of Stuttgart, which are supported by the academic staff of the Institute of Space Systems. The satellite features a camera payload and various scientific experiments. The latter are dedicated to the detection of molecular oxygen and heat-flux measurements in the upper layers of the atmosphere shortly before re-entry [54].

Unlike *EIVE*, the prime goal of the *SOURCE* mission is not just the in-orbit verification of a component, but the education of students in satellite development and operations. Students are trained in all disciplines related to space mission projects, starting with the definition of the project, requirements engineering, the design process, the implementation, and ending with test & verification eventually. The training is also not limited to the development of the satellite, but covers the entire space system, including the space *and* the ground segment. Right from the beginning of the project, students also got involved in the definition of the new operations infrastructure.

Table 2.1: List of IRS Ground Stations

Key	Location / Operator	TC	TM	Data
IRS	Stuttgart (GER) / IRS	✓	✓	✓
WHM	Weilheim (GER) / DLR	✓	✓	
INU	Inuvik (CAN) / DLR	✓	✓	
OHG	O'Higgins (Antarctica) / DLR	✓	✓	
NYA	Ny-Ålesund (NOR) / GFZ		✓	✓
MAL	Kuala Lumpur (MAL) / UPM		✓	✓

**Figure 2.3:** Global Distribution of IRS Ground Station Network

2.2.3 Ground Station Network

As introduced in section 2.1.2, the *Flying Laptop* ground segment already features a series of ground stations. That network is constantly growing. For instance, in the scope of a collaboration with Universiti Putra Malaysia (UPM), a new antenna system was set-up in south-east Asia, adding S/X dual band reception capabilities to the network [16]. All ground stations actively used for the *Flying Laptop* mission are listed in table 2.1. The global distribution of that network is shown in figure 2.3.

The DLR ground stations in Germany, Canada and the Antarctica were only used during LEOP and have been utilized as back-up system for TM/TC since then. A similar use case is considered

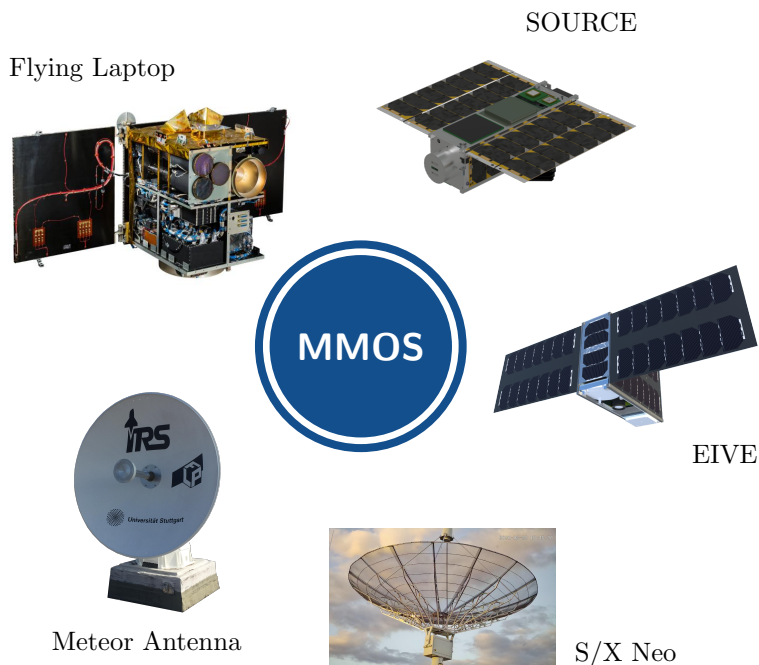


Figure 2.4: Missions to be Operated by the MMOS

for the upcoming missions. The ground station of the GFZ in Ny-Ålesund, Norway is used for TM and payload data downlink. Same applies to the new 4.5 m dish antenna in Malaysia.

For daily operations and TC uplink *Flying Laptop* and all follow-on missions fall back to the university ground station. With the upcoming mission *EIVE*, the number of antennas located at the institute premises is growing too.

2.3 The New Operations System

The multitude of new satellite missions University of Stuttgart is going for raises the demand for a new ground infrastructure and a

new operations system. That new operations system shall be the MMOS (fig. 2.4).

That System not only must be capable of operating multiple satellites simultaneously, it also must be capable of scheduling the different ground stations and antennas in the network. Furthermore, it must deal with the challenge of handling multiple satellites competing for the limited link capabilities of these antennas.

Because, the rework of the existing ground system would have meant an enormous amount of implementation work anyway, the decision was made to start the development of an all new Operating System (OS), which serves the requirements of each upcoming mission equally. This is when the MMOS project was initiated. Starting point for that development were of course the solutions and the lessons learned from the *Flying Laptop* mission.

Objectives

The striven MMOS shall be a system specifically designed for the operations of multiple distributed satellite systems in parallel. The system design shall be contemporary and must therefore take pace with the ongoing evolution of satellite mission operations and shifting operational paradigms, which shall be discussed briefly in the following.

3.1 Paradigm Shifts in Satellite Mission Operations

Since various new space companies have entered the satellite market over the past decades, missions and earth bound constellations are not just projects of big syndicates or governmental agencies anymore. Through a developing industry, the availability of standards such as the CubeSat Design Specification [21], and a growing market of commercial off-the-shelf (COTS) components, even larger fleets become feasible for new and smaller stakeholders in the space sector. As a side effect of that development, conventional operational paradigms have been shifted. Ben-Larbi et. al. have identified the following [11]:

1. Due to the reduced costs and the arising opportunities of being launched as secondary payload, organizations favor small satellites or CubeSats instead of large satellite buses. And since the barriers of launching small satellites are getting fewer, organizations tend to use multiple missions instead of single satellites to achieve operational or scientific goals. Consequently, these organization must be capable of multi-mission operations.
2. The operation of fleets and constellations with limited personnel is not possible without a sound automation concept on ground. New concepts promote automatic operations during all mission phases, even during LEOP, commissioning, and contingency operations. This covers mission planning, the allocation of resources in space and on ground, telemetry monitoring, anomaly reaction etc.
3. “A constellation must be considered holistically [as one system]. Performing detailed monitoring of individual satellites is no longer possible for large constellations and operations techniques are moving towards monitoring and operation of the system as a whole. Metrics must be defined that allow for the overall system health to be assessed.” ([11], p. 42).
4. Classical telemetry/telecommand systems are replaced by Service-Oriented Architectures (SOA). “High level abstractions are created for standard satellite activities. Instead of sending individual telecommands and receiving telemetry confirmation, the system is restructured such that these low-level commands are transparent to the requester. All system processes between the request and the output are automated and require minimal manual interaction.” ([11], p. 43).
5. Instead of dedicated control centers, modern operation systems feature cloud based architectures, allowing operators to connect from anywhere via web front end. This reduces the

complexity of the required infrastructure on site and consequently the demand for hardware maintenance [11].

3.2 Reference Mission Architecture

Designing a mission OS for a distributed satellite system is probably the most vague objective that can be formulated. Before such a system can be implemented, it is necessary to make a step back in order to clarify the key characteristics of a distributed satellite system. After that, a reference mission architecture can be specified, against which a mission OS can be designed.

Constellations as a special kind of a distributed system are usually designed to provide global communication, navigation or Earth Observation (EO) services. The service is thereby improved that a global or local demand is not satisfied by just one, but multiple satellites. So in the first instance, a constellation is nothing more than a distributed system of at least two satellites.

Beyond that, a more precise classification is necessary. G. B. Shaw et. al. classify distributed satellite systems by means of the operating principle, the spacial distribution, the architectural homogeneity and a set of operational characteristics [117]. In terms of the operating principle, Shaw et. al. distinguish between *collaborative* systems, where each satellite contributes independently to an overall mission goal, and *symbiotic* systems, where the distributed system can only function as a whole. An example for a collaborative architecture is Planet's earth imaging constellation *Dove* [11]. A symbiotic system for instance is the German *TanDEM-X* mission providing radiometric EO, where the image receiving satellite cannot function without its illuminating satellite¹ [82]. Regarding the architectural homogeneity, Shaw et. al. distinguish between the homogeneity in the spacial distribution of the satellites and the homogeneity in the used satellite architectures. The latter is achieved,

¹Depending on the operational mode

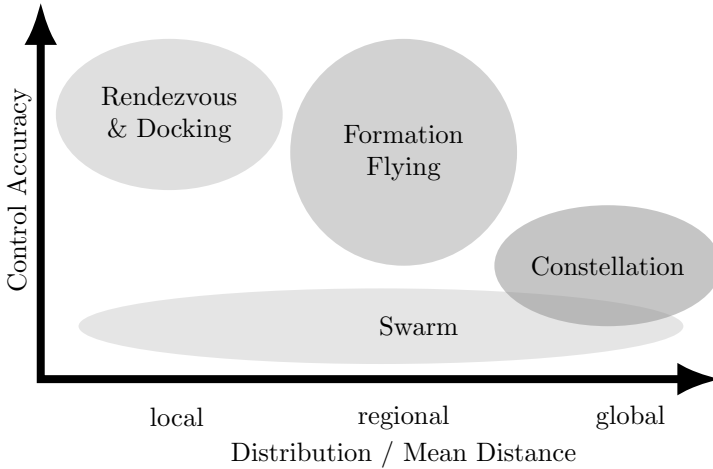


Figure 3.1: Classification of Distributed Satellite Systems [57]

if all satellites in the distributed system are identical. Regarding the homogeneity in the spacial distribution, Shaw et. al. define local *clusters* of satellites flying in close proximity (also known as *formation*), *constellations* of mostly identical satellites distributed on different orbits (usually following a Walker Delta pattern), and a combination of both. Finally, Shaw et. al. consider operational aspects to classify distributed satellite systems. Last cover e.g. sensing methodologies (active/passive), data collecting modes and the availability of the provided service [117].

Inspired by nature, E. Gill introduces a different classification approach (fig. 3.1). Gill classifies distributed satellite systems by the mean distance of the spacecraft, the way how the satellites interact, and the required control accuracy (e.g. for station-keeping). He further defines the *swarm* as a special kind of constellation, consisting of a larger number of low-cost satellites with very limited control capabilities. A characteristic of a swarm is that a single satellite can be taken away from the mission without significantly impairing its

Table 3.1: Example Classification of Distributed Satellite Systems, from [11] - For each mission, the table displays: the number of satellites, the global distribution, the working principle, the homogeneity, and the service availability

GPS	Meteosat	Sentinel	Starlink
<i>31</i>	<i>4</i>	<i>7</i>	<i>240 / 4425</i>
<i>global</i>	<i>regional</i>	<i>global</i>	<i>global</i>
<i>symbiotic</i>	<i>collaborative</i>	<i>collaborative</i>	<i>collaborative</i>
<i>homogeneous</i>	<i>inhomogeneous</i>	<i>inhomogeneous</i>	<i>homogeneous</i>
<i>global</i>	<i>regional</i>	<i>local</i>	<i>global</i>
<i>Constellation</i>	<i>Formation</i>	<i>Constellation</i>	<i>Swarm</i>

functionality [103]. Furthermore, swarm satellites, as in case of the *Dove* mission, usually do not feature inter-satellite communication.

Rendezvous & Docking systems are formed by two pairing satellites. Rendezvous as well as formation flying have in common that they demand for a high level of position control accuracy in the order of 1 m and better [57].

Some examples for the introduced classification approaches are listed in table 3.1.

The consideration of these aspects is necessary, because they directly affect the operational concept. Depending on the kind of the constellation, different requirements emerge regarding the OS scalability, the automation concept, and the mission scheduling approach. Also, boundary conditions for operational optimizations depend on the kind of the constellation. For example, the scheduling of ground station communication times is more problematic if multiple operated spacecraft are located in close proximity to each other.

As long as the operated constellation is not defined in detail, an OS would have to be designed against very uncertain requirements. Considering five of the criteria mentioned above means 216 possible permutations (tab. 3.2). Realizing a generic design from scratch,

Table 3.2: Possible Permutations of Distributed Satellite Systems

Characteristic	Options	Variants
Size (magnitude)	$10^0 / 10^1 / 10^2 / 10^3$	4
Distribution	<i>global / regional / local</i>	3
Principle	<i>symbiotic / collaborative / both</i>	3
Homogeneity	<i>homogeneous / inhomogeneous</i>	2
Service Availability	<i>global / regional / local</i>	3
Permutations:		216

which serves all of them equally in all operational aspects, is not realistic.

Hence, the solution is to start with a baseline implementation of a mission OS, providing a subset of features distributed satellite systems require in general. Nevertheless, even such a baseline OS must be designed and verified against specified criteria, which is why a reference architecture must be defined. Therefore, the following sections discuss the definition of that reference architecture for the MMOS by means of the five key characteristics: system size, spatial distribution, operating principle, architectural homogeneity and service availability.

3.2.1 Size

Figure 3.2 depicts the sizes of active and planned satellite constellations, since the launch of the GPS in 1978. The evaluation is based on a register, created by [11] (appendix. A.1).

Since the beginning of the 21st century, the number of active constellations in orbit has increased significantly. Yet, there is no long-term trend that constellations are also getting larger by orders of magnitude. There are indeed plenty of companies that have announced the launch of constellations significantly larger than 100 or even 1000 satellites during the last decade, yet it is not sure that this trend can be pursued in the face of space debris dissemination

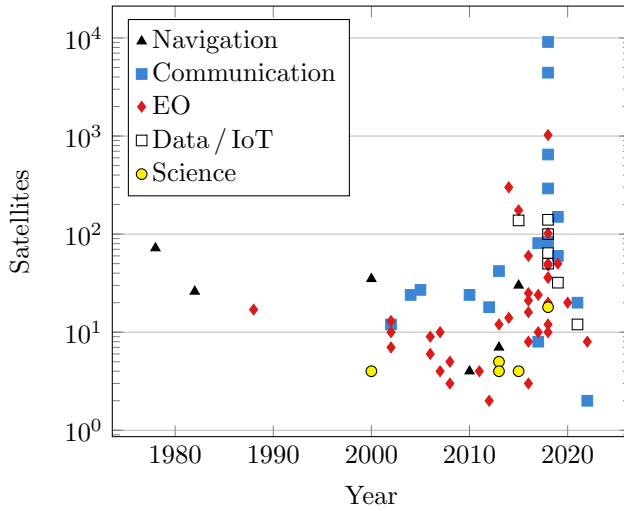


Figure 3.2: Sizes of *active* and *planned* constellations since 1978, broken down by service type - The plot shows the total size of the respective constellations, including active, inactive and planned satellites. The date is the year of the first launched satellite. Data from [11].

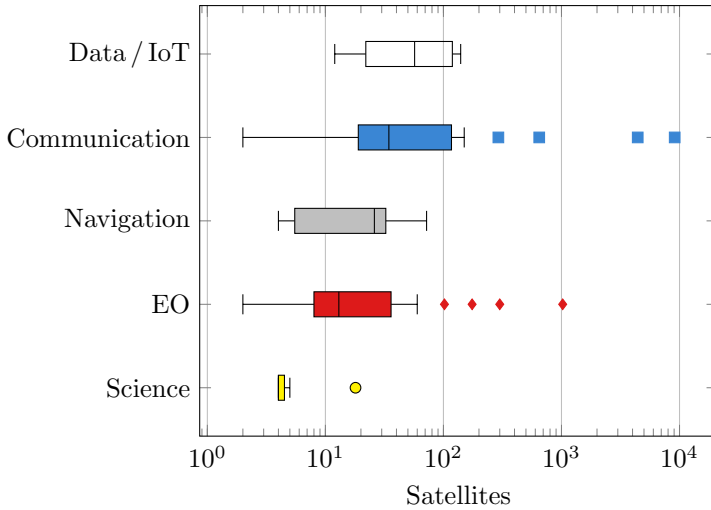


Figure 3.3: Distribution of constellation sizes, broken down by mission type. Data from [11].

and other limiting factors, such as the finite number of usable radio frequency bands ([35], p. 19). Today, the median size of a constellation is 20, and the majority of active constellations is significantly smaller than 100.

Another trend that can be observed is, that the constellation size correlates with the provided service. With a median sizes of 42, respectively 64 satellites, constellations providing communication, data or IOT services are noticeably larger than constellations providing scientific or EO services. The median sizes of the latter are 14 and 4 (fig. 3.3).

Due to the immense scattering of the constellation sizes and the uncertain use case, the envisaged MMOS design and thus the reference mission architecture should be oriented towards the area of application that is most likely. For instance, organizations operating larger fleets for navigation, communication, and data services usually have their own, tailored mission operations solution, a generic baseline mission OS cannot compete with. Thus, from an academic

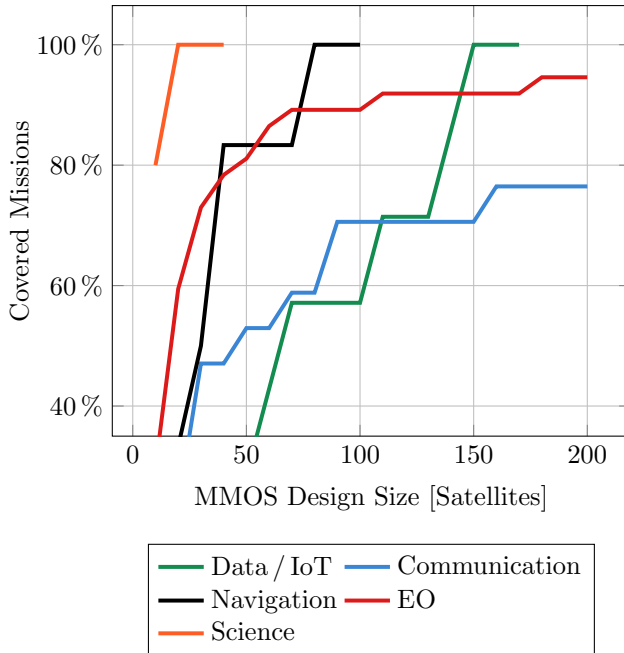


Figure 3.4: Percentage of missions covered by the MMOS design size, broken down by mission type

point of view, the consideration of a science or an EO mission makes more sense.

As mentioned above, the median size of the listed EO missions is 14. With a median size of 4, scientific missions are even significantly smaller (fig. 3.3). Hence, a constellation size in the order of magnitude of tens is a valid assumption for the reference mission. For example, an MMOS designed for a constellations size of 50 satellites would cover about 80 % of the EO missions and all of the scientific missions to date (fig. 3.4).

3.2.2 Spacial Distribution

A number of operational aspects are affected by the spacial distribution of the satellites. Such as

- the scheduling of ground station passes and the resolution of conflicts,
- the necessity for maneuver planning, and station-keeping
- or the level of satellite interactions.

Generally, these issues are getting more problematic the closer the satellites are located to each other. In order to cope with these, a sound constellation OS should therefore be capable of scheduling activities of spacecraft moving in close proximity; in particular because even globally distributed systems are usually deployed as a cluster, before the individual satellites drift into their final orbit position [63].

3.2.3 Operating Principle

Within a collaborative architecture, each satellite performs a task independently from the rest of the constellation and the malfunction of a single satellite does not necessarily impair the mission goal. A symbiotic system instead is characterized by the fact, that each individual satellite only performs a subset of a mission task. Thus, the mission can only function as a whole, if each individual satellite performs correctly [117].

Operating a symbiotic system can mean that activities of different satellites must be coordinated. This coordination (e.g. by a mission planning software) is an extra effort not necessary for a collaborative system.

Consequently, a baseline constellation OS should support the handling of a symbiotic system, and thus the association of various satellite sub-tasks with an overall task.

3.2.4 Architectural Homogeneity

Ground segment and space segment of a mission are the two sides of the same coin, which shall mean that the mission as a whole

can only function if ground segment architecture and space segment architecture are compatible to each other. Compatibility can thereby be achieved, that both parties agree on a common implementation of several aspects, including but not limited to:

1. Physical Communication Layer (frequency band, polarization, modulation, etc.)
2. Security Mechanisms (e.g. encryption mechanisms)
3. Data Abstraction (e.g. according to CCSDS [23] or OSI [74])
4. Protocols
5. TM, TCs, and parameter definitions etc.
6. System Topology

The physical communication layer to be implemented by the satellite is usually a boundary condition coming from the mission design. Yet, it is the matter of the used ground stations and not of the mission OS as described here to cope with the physical layer. Security mechanisms such as encryption methodologies are another sparkling source for compatibility problems between spacecraft, ground station and OS, but shall not be discussed in-depth hereinafter too. What will be discussed is the way data is abstracted by the space system and how this is reflected by the mission OS. Furthermore, OS and space system must agree on the same protocols, and a common set of TM, TC and parameter definitions. In other words, mission OS and spacecraft must speak the same language. The dictionary for that language is what is commonly known as Satellite Reference Database (SRDB) or Mission Information Base (MIB) in European spacecraft operations.

Furthermore, the satellite topology can also have a significant impact on the design and the working concept of a mission OS. The MMOS, as will be discussed later on, is designed for a hierarchical satellite topology, featuring a central OBC, which controls the

spacecraft. There is no proof, that the concept of the MMOS also applies to satellite topologies that are fundamentally different. For instance, the satellite itself could consist of a federal bus system with a number of distributed control entities ([134], p. 610 f.).

In case of an inhomogeneous constellation (non-identical satellites), each individual satellite design must be compatible with the OS.

But even in case of a homogeneous constellation (identical satellites), individual satellites can deviate from the actual design. These deviations can be the result of bug-fixes, workarounds, different software versions or the rollout of new satellite generations. Because of that and because of the mutual dependencies listed above, each satellite within a constellation should be registered, mapped and commanded individually by the MMOS. That means that each mission must be regarded as inhomogeneous in terms of the satellite architectures, even though the constellation is homogeneous by design.

3.2.5 Service Availability

Distributed systems providing a global service, such as GPS or Starlink (tab. 3.1), usually perform the same task again and again, 24/7. This means that the fulfillment of the pure mission task, e.g. the delivery of position data by the GPS, does not require a tailored activity scheduling from ground.

Providing a local or a regional service instead, e.g. the mapping of an area of interest (AOI), usually requires the individual scheduling of spacecraft activities. Such activities are time and location dependent, because they are tailored according to individual customer needs. This is an additional effort, compared to the operations of space system, working autonomously 24/7.

Table 3.3: Characteristics of the Reference Architecture for Requirement Generation and the Derivation of System Features

Characteristic	Selected
Size (magnitude)	→ 10 ¹
Distribution	→ local
Principle	→ symbiotic
Homogeneity	→ inhomogeneous
Service Availability	→ local/regional

A baseline mission OS, should support and individual, time, and location dependent activity scheduling though. This is why the M-MOS targets missions providing a local or regional service.

3.3 Summary

Table 3.3 summarizes the characteristics of the satellite system reference architecture against which the MMOS shall be designed. All non-functional and functional requirements as well as all system features are derived from that architecture, or are formulated accordingly.

The considered system size is about the order of ten, according to the estimated demand for scientific or EO missions in the academic sector. The satellites are locally distributed within a formation, which results in the demand for more sophisticated maneuver scheduling or pass planing capabilities by the ground system. The selected operating principle of the satellite system is symbiotic, meaning that all of the spacecraft perform a subtask of an overall goal and that the system can only function as a whole. In other words, each satellite performs an individual activity and the overall goal can only be achieved, if individual activities are executed successfully. In terms of the architectural homogeneity, an inhomogeneous design was selected for the reference architecture. This means that the satellites within the constellation can be implemented vari-

ously. Finally, it is assumed that the reference architecture provides a local/regional service.

Because, the envisaged MMOS is supposed to be a generic baseline implementation of a constellation OS, each of the considered characteristics has been shifted towards the use case that was considered most universal. This is why the introduced sketch of a reference architecture is more a theoretical, worst case scenario than a realistic mission that will be operated.

3.3.1 Top-Level Requirements

From the previous considerations a number of top-level requirements can be derived:

- *Single Satellite Operations* – The MMOS shall be capable of operating a single satellite mission, where the term *operations* is understood as the sum of all activities to monitor and to control the *operated* satellite, and to execute the mission.
- *Multi Satellite Operations* – The MMOS shall be capable of operating multiple satellite missions simultaneously.
- *Ground Station Operations* – The MMOS shall be capable of operating ground stations. The ground station schedules shall be made available to all the other missions for their planning process.
- *Constellation Operations* – The MMOS shall be capable of operating a constellation as described in section 3.2.
- *Scalability* – The MMOS shall allow the adding of new missions into the working system.
- *Mission Interference* – The various mission operations processes must not interfere with each other.

- *Automation* – The MMOS shall implement a control concept that allows for automatic operations.
- *Baseline Features* – The MMOS shall provide all baseline features of a satellite operations system, so that a satellite system can be monitored, scheduled, and commanded without the need of a third party solution, or the need of a rework. Missions specific behavior of baseline components must be achievable through configuration only.
- *Telemetry and Telecommand* – The MMOS shall be capable of exchanging TM and TC with the operated satellites.
- *Archive* – The MMOS shall feature a central telemetry archive for all its operated systems.
- *Flight Dynamics* – The MMOS shall feature a flight dynamics system capable of orbit determination, orbit prediction, and the determination of contact times with the satellites. The orbit prediction shall further consider boundary conditions such as environmental effects, and scheduled maneuvers.
- *Planning* – The MMOS shall feature a planning concept capable of resolving the state of the operated system. It shall further be possible to schedule activities and to resolve conflicts between them. That scheduling concept shall not make a restriction in terms of the type of operated system.
- *Interaction* – The scheduling concept shall allow for the symbiotic collaboration of operated systems, the coordination of activities across system boundaries, and thus for the planning of system interaction.
- *Extendability* – The MMOS shall feature a set of standardized interfaces, which allow the integration of mission specific software components into the operations process.

The requirements above cover the pure operational functionality of the MMOS, which is referred to in the following. Beyond that, the system itself needs to be set-up, configured and maintained. This means additional requirements like the demand for a user management, an automatic orchestration, an internal component logging, security concepts, etc. These requirements are not listed here, because those topics cannot be dealt with conclusively in this work.

Methodology

Purpose of any software is to utilize a computer architecture for the solution of a certain task. In this case, a software for satellite operations is required, which needed to be developed from scratch.

This section guides through a series of methodologies, which have been applied on the way towards this particular software, beginning with the problem of requirement identification and ending with methodologies for testing and verification of the product. Following a top-down approach, this chapter starts with a global, macroscopic view of the problem and subsequently goes further into details of the solution.

4.1 The Wife and the Mother-in-Law

In the year 1915, cartoonist William E. Hill posted an illusion in *Puck*, a former American humor magazine. The illusion is shown in figure 4.1. It was titled “My wife and my mother-in-law. They are both in this picture - find them” [65]. Nowadays, this image is shown on many occasions to demonstrate that people can see different things in one and the same image. Depending on the chosen perspective, the picture either shows an old lady with a big hooky



Figure 4.1: *My Wife and my Mother-in-Law* by William E. Hill - 1915 [65]

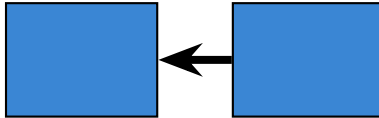


Figure 4.2: Two Boxes and an Arrow

nose, or a young woman with her head turned away from the observer.

The picture shall relate to a well known problem. When a group of people works together on a technical problem, figures and graphs are always a good means of discussing certain aspects of the solution. However, depending on the background of the members of the group, these figures are usually interpreted in at least as many ways as people are involved. This shall be demonstrated by figure 4.2.

Figure 4.2 shows two boxes, which are linked by an arrow pointing towards the left one. The questions is: What could people see in that picture? Depending on the type of person, answers can be manifold. A person with a non-technical background would probably answer: “I see two blue rectangles and an arrow in the middle.”

A system engineer would maybe answer: “I see two systems and the right system provides information to the left system.” And an IT-specialist would possibly say something like: “I see a data channel between two communication partners, where the right one subscribes to the left one.”

If it is not stated perfectly clear what a figure is supposed to express, those various possible interpretations can lead to long and extensive discussions during the engineering process. The problem with this kind of discussion is often that at the end of the day two persons leave the room, where one of them thinks to be right and the other one thinks to be proven wrong, although that has never been the case. A solution to this is an agreement on standardized forms of display. If for instance figure 4.2 would really display two communicating systems, then it must be specified somewhere what the black arrow means. It could be a network connection, a data flow, or some sort of functional interface.

This particular problem will appear and be discussed on many occasions within this work. Therefore, consistent forms of display are applied in this work. If not stated otherwise, a technical implementation is always displayed in the same manner, as well as one form of display always refers to the same kind of technical implementation.

4.2 Project Organization

Whenever a new project is initiated, the question that arises is the one about how the project shall be organized. Books are written about that challenge and an answer is not easy to find, as the best way depends on a series of boundary conditions. The trade-off is generally made between a fully process oriented approach and an agile development, where many favor the agile approach, simply because it is modern these days. In her book about Agility within ISO 9001, P. Adam mentions that some enthusiasts think of agility as the

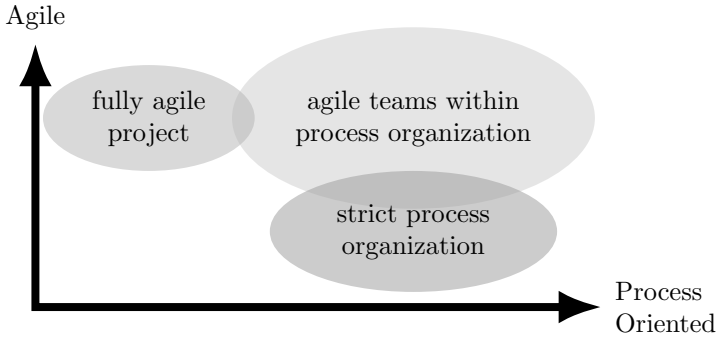


Figure 4.3: Agile vs. Process Oriented Approach, following [3]

one and only way of organizing a project ([3], p. 15), but apparently, it is not that easy.

4.2.1 Agile vs. Process Oriented

The selection of a suitable way of project organization depends on a series of boundary conditions, such as the interfaces to the environment organization, the availability of competences and eventually the ability of the team members to organize themselves ([3], p. 35 ff.). An organization or a project does not necessarily have to follow either the one or the other approach. Hybrid solutions exist too, as figure 4.3 illustrates.

At project initiation, the members of the team responsible for the Multi-Mission Operations System (MMOS) project faced a series of challenges. The most prominent problem at project launch was the work load of each team member that time. As mentioned in chapter 2, Institute of Space Systems had gone through the realization of a series of space mission projects. As it turned out, almost every team member had part-time obligations within at least one of these projects, which limited the available man-hours and made the planning complicated.

The limited team size directly led to a lack of capacity for team lead, project management and planning. As it turned out after the first weeks, consequently tracking the entire project state and updating the project plan simply was not feasible. That was not just because of the limited man-hours but also due the uncertain and changing boundary conditions of the project. When the project was officially initiated, a primary objective needed to be specified in order to get everyone on the same page regarding the goal and the extend of the project. The problem was though, some of the missions which demanded for an operations systems had not specified their concept of operations yet. So the project organization needed to be capable of reacting on changing user requirements.

All of these practical aspects directly spoke in favor of an agile development approach. The demand for a methodology to cope with such a volatile, complex and uncertain environment is one of the reasons why agile methods exist in the first place [3].

4.2.2 Agile Methods

While supporters of agility claim that everything must be agile, opponents think that this sort of project organization only results in chaos [3]. To prevent this from happening, a series of agile methods exist to structure the work flow. One of the most prominent ones is *Scrum*, which was developed in the early 1990s by Ken Schwaber and Jeff Sutherland [115]. Meanwhile the approach is well established in the world of software development.

The principle idea is that a rather small team works independently on a predefined goal in an iterative process. The advantage of this methodology is that intermediate results (increments) are generated on a frequent basis, and that these increments can be constantly assessed and compared to the user requirements. In this manner the approach enables the team to react agilely on changing customer needs or varying boundary conditions.

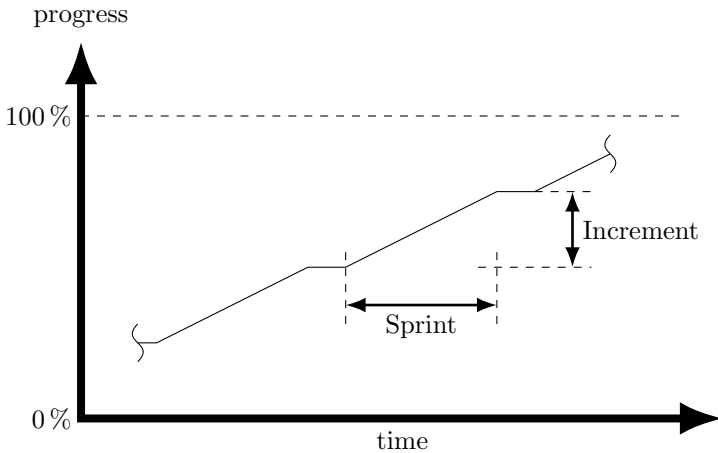


Figure 4.4: Progress of Scrum

Within Scrum the product is developed during consecutive working periods, called *sprints*. Outcome of each sprint is the *increment*. Prior to each sprint, the team plans the working period by defining the target increment, based on the current product backlog. The backlog is a dynamic document, listing the requirements on the envisaged product. This document can change and is updated constantly, since the requirements on the product can change.

According to *The Scrum Guide* [115], the duration of a sprint is usually a month, depending on the size of the increment. During a sprint, the work progress is constantly monitored by means of short daily meetings, called *Daily Scrums*. These meetings shall make the progress transparent to everyone and indicate delays. A Sprint terminates after the work on the increment is done, which requires that the extend of the work has been defined specifically in advance.

After a Sprint, the work of the development team is reviewed. That review is conducted by the development team and potential stakeholders or customers. Purpose of this review is to identify what has been done, which problems appeared, and to clarify what needs to be done next. The outcome of the review is documented in the

product backlog. A sprint review is followed by the complementary *sprint retrospective*, a meeting that shall ensure the productivity of the team and support the further planning [115]. The process is illustrated by figure 4.4.

A few roles within the team are defined by the Scrum Guide. The first one is the *product owner*, a person in charge of documenting the state of the developed product. That person must ensure that all needed information about the product is available to the rest of the team, which covers the backlog and the definition of the sprint goals. The second group is the *development team*, which is a self-organizing unit responsible for the delivery of the increment. The last one is the so called *Scrum master*, an experienced associate who takes care that the concept of Scrum is clear to everyone and that the rules are followed.

4.2.2.1 Applied Methods

Schwaber and Sutherland think of Scrum as a framework, rather than a defined method or a technique. They claim that institutions are free to implement their own “tactics for using the Scrum framework” ([115], p. 3).

This section introduces some of the agile methods that have been applied in the scope of the project.

Technical Meetings

Especially during the very early stages of the project, technical meetings were a good means of discussing a specific topic or problem in order to gain domain knowledge, or to shape the design of the MM-OS architecture.

Central element of these discussion was usually a preliminary design, and/or a whiteboard, which the participants used to develop ideas and thoughts. These whiteboard sketches were photographed afterwards, so that everyone was able to recall what has been discussed. The value of these photographs was very limited though.

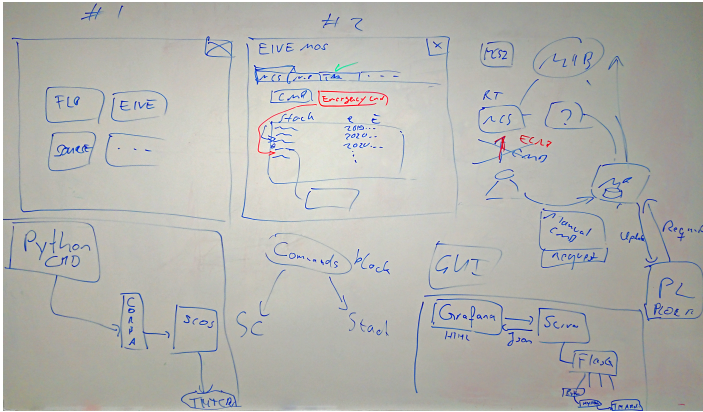


Figure 4.5: Whiteboard Sketch (Example)

For instance, a sketch as the one depicted in figure 4.5, does not enable someone external to follow the discussion. Even for participants these sketches become worthless eventually, because the respective trains of thought vanish or the findings become obsolete. This is why the discussed content was formally documented right after the meeting. The created documents could then be referenced and used in the subsequent development process.

At the beginning of the project, technical meetings were held with the entire development team. After assigning responsibilities to certain members of the group, the circle of people participating in such meetings shrank. Limiting the number of participant to three or four significantly increased the efficiency of those meetings.

Design Backlog and *MoSCoW* Prioritization

The design backlog has been implemented as a means of recording the requirements of a certain system component after it had been further decomposed into various functional elements. A short extract from the backlog of the MMOS Mission Planning Tool (MPT) is shown in figure 4.6. For the complete MPT backlog, it is referred to appendix B.1.

Status	Date	Key	Name	Description	Parent	Priority	Verif.
To Be Done	16.03.2021	MPT.5	Configuration Layer	The MPT shall have configuration (CONF) layer. Purpose of the CONF layer is to initialize the MPT, to load the mission specific configuration of the tool and to gather information about the status of the tool.	none	Should	Review
In Progress	16.03.2021	MPT.COM.1	Gateway	The communication layer shall implement a gateway to connect to the Middleware	MPT.1	Must	Inspection
In Progress	16.03.2021	MPT.COM.G.1	Routing	The Gateway shall be capable of directing messages to a specific component in the System	MPT.COM.1	Must	Test
In Progress	16.03.2021	MPT.COM.G.2	Reception	The Gateway shall be capable of recognizing messages from a different component in the system and provide the content to the respective feature in the MPT.	MPT.COM.1	Must	Test
In Progress	16.03.2021	MPT.COM.G.3	Downtime	After a down-time of the MPT, the gateway shall be capable of collecting messages, buffered by the Middleware and provide them the target features in the MPT.	MPT.COM.1	Should	Test
In Progress	16.03.2021	MPT.IFL.1	Activity & Schedule Interface	The MPT shall have an interface to access the mission schedule, which is stored in a central database That IF shall be called the Activity & Schedule Interface (ASI)	MPT.2	Must	Inspection
To Be Done	16.03.2021	MPT.IFL.2	Phase Interface	The MPT shall have an interface to make phase requests at the FDT. That IF shall be called the Phase Interface (PIF)	MPT.2	Should	Inspection
To Be Done	16.03.2021	MPT.IFL.3	Data Interface	The MPT shall have an interface to connect to the central system TM Archive. That IF shall be called the Data Interface (DIF)	MPT.2	Should	Inspection
To Be Done	30.04.2021	MPT.IFL.4	Monitoring Interface	The MPT shall have an interface that allows other (system management) instances to monitor the state of the MPT as an instance. That IF shall be called the Monitoring Interface (MIF)	MPT.2	Could	Inspection
To Be Done	30.04.2021	MPT.IFL.5	Control Interface	The MPT shall have an interface that allows other (system management) instances to control the MPT as an instance. That IF shall be called the Control Interface (CIF)	MPT.2	Could	Inspection
To Be Done	30.04.2021	MPT.IFL.6	MIB Interface	The MPT shall have an interface that allows mission specific configurations from the MIB into the MPT. That interface shall be called MIB Interface (MIBIF)	MPT.2	Wont	Inspection

Figure 4.6: Extract from Backlog of an MMOS Component

Within a backlog, requirements are hierarchically linked by means of a key and a parent-child referencing. This ensures that the entire functional design of a component can be mapped in the log. A status flag helps keeping track of the implementation state of each requirement.

For a better definition of design increments, each requirement is prioritized. In this case, the *MoSCoW* Prioritization scheme was applied ([84], p. 90).

Increment Definition

Guided by the prioritization scheme in the backlog, increments for a sprint can be defined.

An increment is characterized by a number of features or functionalities, which have a particular value for a user of the component. These features are formulated in so called *epics*, where an epic describes the implemented functionality from a user's perspective, like in the following example:

1. The user can select a command from a drop-down list.
2. The user can manually edit the command parameters.
3. The user can hit a button to add the command to the command stack.
4. The user can hit a *send* button for command release.
5. The user can see acknowledgment information.

Each epic further references the requirements, which it implements. This way, the result of the increment can be traced in the backlog.

4.3 Domain Engineering

With the problem and requirements being specified, a software system architecture can be designed.

Again, purpose of the project is the development of a system for multi-mission and constellation operations. Ignoring different naming conventions, Operating Systems (OS) usually consist of always the same functional elements namely: flight dynamics, mission planning, ground data handling, and so on. ([134], p. 905, fig. 29-1). Assuming that all these systems are well understood, why is making them multi-mission capable so complicated?

What has been identified quite soon is that most OS are tailored solutions, and that the key to a multi-mission application is *reuse*. Thus, a maximum portion of the MMOS software must either be used commonly, or, if that is not possible, instantiated several times and then *configured* for a respective mission.

Software components instantiated once and used commonly by all missions are referred to as *common* elements within the MMOS. Components, which are instantiated and configured separately for each mission are referred to as *generic* elements. A quality of generic components is that it always executes the same source code and that instances only differ by their configuration.

Of course, satellite operations, especially automatic operations, always falls back on mission specific solutions for certain tasks. These components need to be developed from scratch for each new mission. The MMOS must therefore provide standardized interfaces to allow the integration of such customized items into the architecture.

To distinguish uniquely between common, generic, and mission specific components, this work applies a colour scheme, which is shown in figure 4.7.

However, the idea of composing an architecture from reusable components causes some problems. The first one is the problem of orchestration. Instantiating a variable number of generic and mission specific elements in a working environment is a non-trivial process.

The second one is a problem of software quality. As a matter of fact, it cannot be guaranteed that an object tested and verified for

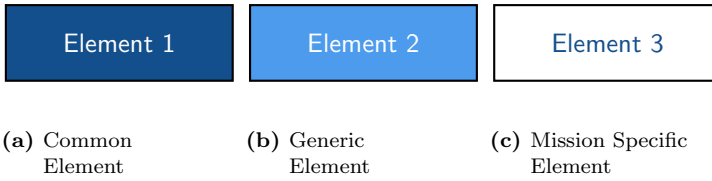


Figure 4.7: Color Scheme for the Identification of the Different Types of Elements

a specific use case also performs correctly in another use case ([121], p. 22).

And finally, the third problem is the one of answering the questions: Which components in the architecture can be reused? The discipline of answering that question is called *domain engineering*. Domain engineering is the discipline of developing reusable software, where the term *domain* specifies

“[...] an area of knowledge [...] including a set of concepts and terminology [...], and including knowledge of how to build software systems [...] in that area” ([31], p. 41).

The concept dates back to the doctoral thesis by James M. Neighbors [94], who in 1980 identified the problem that the major portion of the development efforts to computing systems goes into the development of software rather than the hardware. Yet, the lack of capable software development methods back then resulted in what Neighbors calls the *software crisis*. Consequently, he introduced a methodology for organizing the software development process through the identification of similar, reusable elements. This enabled the adaption of existing solutions to various kinds of implementations, where former object-oriented methods only aimed for specific solutions to specific problems [31].

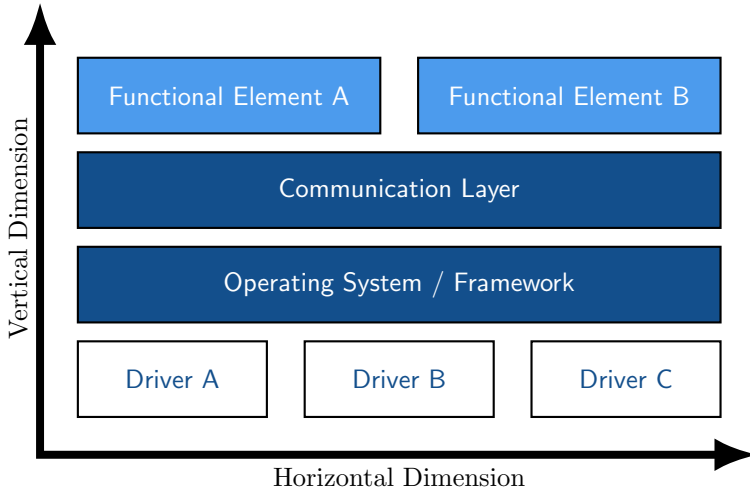


Figure 4.8: Example Software System with two Dimensions

A domain engineering process can happen in two different dimensions, the horizontal dimension and the vertical dimension¹ ([31], p. 34). The idea is illustrated by means of an example software architecture in figure 4.8.

Horizontal domain engineering aims for the development of reusable functional elements. An example will be the Mission Planning Tool. As will be shown in the later course of this work, the Mission Planning Tool in the MMOS is implemented as a generic piece of software, which can be instantiated and configured for an individual mission. The OS then can host and run as many Mission Planning Tools as missions are operated.

Vertical domain engineering however aims for reusable solutions, which apply to the entire system, such as frameworks, interfaces or communication solutions. In the specific case of the MMOS, the outcome of the vertical domain engineering has been a message-oriented

¹Actually, [31] speaks of vertical and horizontal *domain* rather than dimension. Yet, in this context the term *dimension* is preferred, because it is considered more vivid.

middleware that all software components use for information exchange.

The process of domain engineering consists of two consecutive steps: the *domain analysis* and the *domain design* [31], which are introduced in the following.

4.3.1 Domain Analysis

A domain analysis comprises those activities to get from existing knowledge about an area, the *domain knowledge*, to a model of the domain, the *domain model*. K. Czarnecki describes that process not just as a gathering of data, but as a “creative activity” ([31], p. 35). Organizations are encouraged to use creative ways to extend the existing knowledge on the domain. “The sources of domain information include existing systems in the domain, domain experts, system handbooks, textbooks, prototyping, experiments, already known requirements on future systems, etc.” ([31], p. 35). Hence, the process of domain analysis is not a closed procedure, but an iterative activity that refines the domain model as long as the organization gains domain knowledge.

The activities actually comprise a series of processes, such as the characterization of the domain, the collection of data, data analysis and modularisation, and classification [6]. Each process consists of further steps, which are not introduced here, but which can be followed from [31]. For instance, the result can be a hierarchical *feature model* of a system, like the one depicted in figure 4.9.

A feature model is a means of decomposing a technical system into functional elements (features), to identify reusable elements, and to indicate dependencies between those elements. It can also be used to map a real technical device into a software architecture. An example can be found in [9].

Goal of this work is the mapping of all operated systems and the state of those systems within the MMOS. A similar idea is followed

by the *European Ground System – Common Core*, currently developed by ESA, various national space agencies and industry [131]. The mapping of complex systems on ground requires a decomposition of these systems. Hence, all operated satellites, or constellations must be split into individual systems and subsystems potentially. Each identified system can then be represented as an entity by a (reusable) software component on ground.

A result from this domain analysis was that mission operations is not achieved by just mapping the state of the operated satellite. Instead, every involved system including ground stations, antennas, and parts of the MMOS itself must be considered by the feature modelling. The result of this decomposition is discussed in section 5.3.3.1.

4.3.1.1 Feature Model

As mentioned above, domain analysis describes the process of transferring existing domain knowledge into domain models. One possible outcome of this are so called *feature models*. Within domain analysis in general and within Software Product Line (SPL)² development in particular, feature models are a common means of describing the possible functional extend of a system and its derivatives. In this context,

“a feature is a prominent or distinctive user-visible aspect, quality or characteristic of a [...] system.” ([4], p. 324)

An exemplary feature model of an arbitrary spacecraft system is shown in figure 4.9. Feature models are usually displayed in a tree-like structure, called *feature tree*. Each element, called node or product, represents a feature of the system, where each feature (parent),

²SPL: ”systematic reuse of software artifacts across a very large number of similar products” ([97], p. 1)

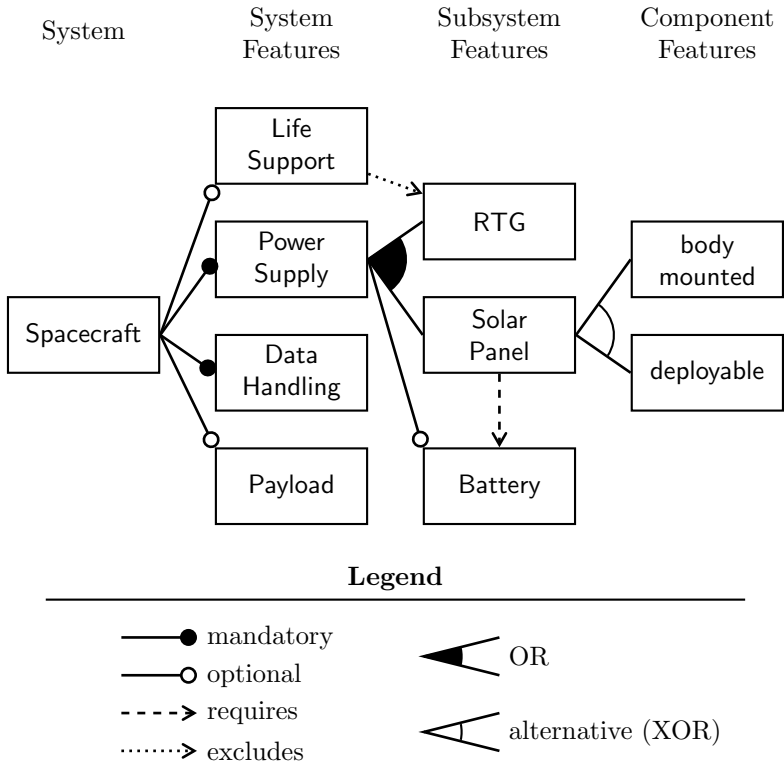


Figure 4.9: Exemplary Domain Feature Model of a Hierarchical System

Table 4.1: Feature Model Relationships

Relationship	Description
mandatory	A product must provide that specific feature
optional	A product can optionally provide that feature
OR	The product must provide at least one of these features
alternative (XOR)	The product must provide exactly one of these features

can have a number of sub-features (children). The connections indicate the logical relationship between parents and their children [107, 97]. They are explained in table 4.1.

Besides the logical relationships, feature models can define constraints between elements. A *require* constraint indicates a feature that is required by a different one, where an *exclude* constraint indicates a feature that is excluded by a another.

At the start of each software development process stands the conversion of the stakeholders' textual descriptions of use cases (sec. 4.2.2) into sound system requirements. According to [34], a requirement is

“a statement that identifies a product or process operational, functional, or design characteristic or constraint, which is unambiguous, testable or measurable, and necessary for product or process acceptability.” ([34], p. 7)

In the easiest form of Requirement Engineering, requirements are simply captured in tables ([34], p. 7) or the product backlog ([115], p. 15). This however shows some drawbacks.

The problem already begins with the user stories. The formulation of a requirement based on a user story can be compared with the translation from one language into another. Like any other in-

terpretation, this holds the risk of information being mistaken if not being missed entirely. In other words, one cannot be sure that the extracted requirements are resilient, and that they really cover the exact scope of the project.

Feature trees can also be a means of solving these problems as they allow for the display of system requirements in a hierarchical and structured manner [97]. This brings a series of benefits, not only with regard to system requirements.

For instance, each feature in the tree is supposed to represent a specific requirement. By hierarchically structuring and linking them, they can be checked in terms of completeness and validity. Unnecessary, and/or redundant requirements become evident right away and conflicts can be made transparent. Organizing features in a hierarchical tree structure also helps defining the scope of a product or a system. This automatically defines the scope of the system testing too, as it supports the definition and the generation of test procedures for a product [97].

As a means of displaying all possible system derivatives, feature trees are a good means of identifying reusable elements eventually. In both the horizontal and vertical dimension, following [31], the methodology allows for the identification of all common, generic, and potentially mission specific features (fig. 4.7).

4.3.2 Domain Design

Scope of the domain design is the transformation of the domain model into an architecture. F. Buschmann et. al. define a software architecture as

“[...] a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non-

functional properties of a software system. [...]” ([20], p. 384)

In that context P. Kruchten’s paper on the *4 + 1 View Model of Software Architecture* [83] is usually cited. In his work, Kruchten addresses the problem that a good architecture must provide enough views, so that each stakeholder of the software can extract the relevant information. [31] compares software architecture with the architecture of a building, which must feature certain information too. For example, a plumber cannot work with an electrical circuit plan.

This is exactly the *Mother-in-Law Problem*, introduced in section 4.1. Consequently, Kruchten has identified a series of stakeholders, which must be considered in the scope of the design process. These are the end users, the programmers, the system integrators, and the system engineers. For each group he defines a specific view and a subset of suitable forms of display which shall not further discussed here.

With a software architecture being defined, a process of identifying recurring arrangements (so called patterns) can be initiated. Patterns aim for the extraction of reusable software parts to form a solution for a particular problem. Emerging patterns can depend e.g. on the type of application, or the design [20].

4.4 Implementation

Compared to the previous steps of the software development, the implementation should be a rather unspectacular process. It basically covers the translation of the system design, usually realized by means of modeling languages, into a coding language, such as C++, Java, Python etc.

The more detailed the architecture has been designed, the better it can be implemented. This does not mean that code generation is a trivial process. Like during house construction based on the architect’s plans, the derivation of software code from a system design

requires certain “skills, creativity and discipline” ([61], p. 109) by the people writing the code. Like an electrician who must be capable of reading a circuit plan, a programmer must be capable of reading a modeling language. And that person must have the technical know-how of converting the content into a specific coding language, like a mason must know how to build a wall according to the construction plans. Such a procedure is of course a sparkling source for human error, which is why software testing is that important (sec. 4.5). It is also beneficial if the person in charge of the software has a certain domain knowledge, so that flaws in the design can be identified during the implementation already ([121], p. 164).

The way towards a compiled piece of software is covered with numerous obstacles and decisions to be made. It begins with the selection of a capable coding language and continues well past the distribution of the source code or the binaries. Yet, this work does not claim to provide a comprehensive discussion of these issues and refers to common literature instead. This is due to the fact that solving problems during software implementation is well discussed, fairly generic and not the purpose of this work in particular.

In the following, those aspects of software implementation are discussed, which are considered the most critical in the context of this work.

4.4.1 Selection of a Programming Language

The choice of a programming language is surely the most important decision in the implementation process. It depends on a series of boundary conditions, the application, and the capabilities of the programmers. [111] has identified ten criteria for the selection of a programming language for distributed systems: The list is largely according to a study of [28], who published an evaluation of the mostly used programming languages between the years 1993 and 2003. In order to estimate the success of a language, the authors

Concurrency	Efficiency	High Integrity
Maintainability	Portability	Reliability
Reusability	Scalability	Security
	Simplicity	

distinguish between what they call *intrinsic factors* and *extrinsic factors*.

Intrinsic factors, such as the ones above, describe the hard technical capabilities of a language. They can be considered to narrow down the number of suitable languages for the job. Extrinsic factors cover aspects such as the availability of support and documentation, the existing knowledge about the language and the number of people using it. If the decision is to be made between a limited number of suitable programming languages, extrinsic factors usually make the difference.

If programmers have to choose between a handful of suitable languages they would most likely use the one they know or like best. In turn, an organization should select a language that is widely used among its programmers. What sense would it make to use a language that nobody else speaks?

Another key aspect is the availability of support and existing knowledge about a language outside the organization. A practical measure of quantifying the dissemination of a language are the statistics provided by online forums such as *StackOverflow* or *Stack-Exchange*. A respective study was performed by [119], who evaluated the number of questions asked about different software frameworks for web page applications, in order to quantify their popularity. Such an evaluation can make sense if the decision is to be made between two or three equivalent languages or within fast evolving domains.

In conclusion, extrinsic factors can speak in favor of a certain language, although others maybe more modern or better in technical terms. Within this project languages were chosen the development

team was most comfortable with.

The selection of a programming language has to be made after the design process and prior to the implementation. But unconsciously, the design process can be already driven towards a certain programming language or framework. This happens, when the people programming the software are involved in the design process.

Like any other language, a programming language is an expression of thought [29, 121]. If programmers are involved in the design process, they usually think about the implementation already and how they would implement a specific idea in their favorite programming language. If a problem occurs that does not fit to a particular language or framework of choice, usually the design is reworked instead of questioning the selected programming approach. This is based on personal observations made during meetings whenever aspects of the system design were discussed (sec. 4.2.2).

To avoid a large variety of coding languages within the project, only a few were allowed eventually:

- C++, and Python for component and back-end programming
- HTML, CSS, JavaScript, and Python (Flask) for user front-end programming.

Further used markup, and database querying languages are not mentioned.

4.4.2 Source-Control Management

Nowadays, there is no serious software project ongoing that is not in some way administrated by a Source-Control Management (SCM), such as *Git*, *SVN* or *CVS*. Main purpose of such a system is to keep track of the versions of the software source code and thus to enable a programmer to return to a stable version of the software in case something goes wrong. Apart from this, the benefits are manifold.

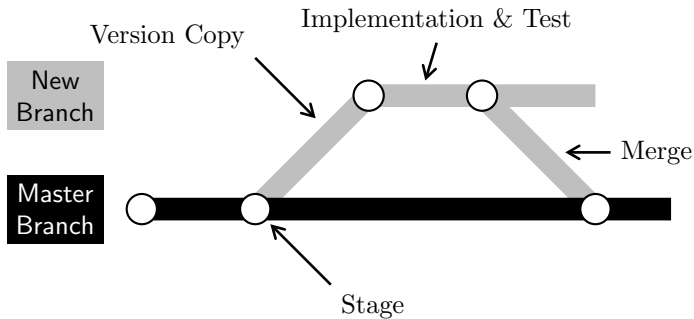


Figure 4.10: Exemplary Version Tree

Git was chosen for this project, since it is widely used in the Linux community and by other organizations such as *Google*, *Microsoft* or *Qt* [116]. The Git project was launched in 2005 and the software is developed further ever since [26].

Within Git, the source code is stored in so called *repositories* on servers. Whenever someone needs a version of the software, the source code can be downloaded (checked out) from the repository. In turn, modifications that shall be published have to be pushed to the repository by the developer.

A great advantage of Git compared to other SCM systems is the concept of *branches*. It is briefly illustrated by figure 4.10. In this concept the software versions (stages) are mapped in a tree-like structure. The main branch is the so called *Master Branch*. Whenever the software has to be modified, a new branch is derived from the Master Branch (or any other parent branch). Modifications in one branch are tracked independently from the others. When the work in one branch is finalized, it is usually merged with the Master Branch again (or its respective parent branch). The concept enables multiple programmers to collaborate in one single software project.

Through hooking of stable branches to automatic tool chains on build servers, SCM also supports the delivery of the product. Stable

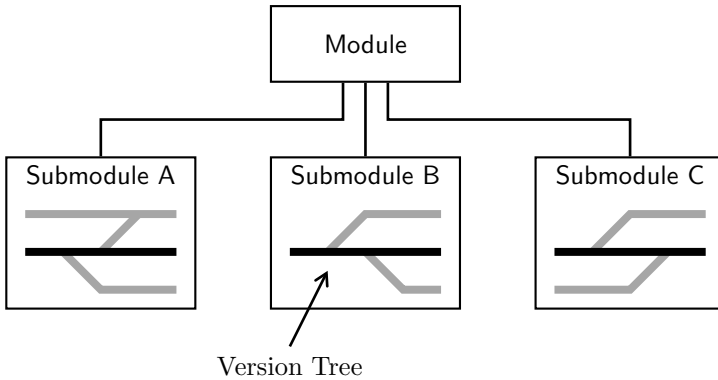


Figure 4.11: Concept of Submodules

branches refer to those, which contain source code that was formerly tested.

Another benefit is the possibility to integrate so called *submodules* into the project (fig. 4.11). Submodules can be separate programs or pieces of software. The difference to branches is that submodules are self-contained projects, with their own repository and their own Git version tree.

Git repositories are usually managed by means of graphical web front ends, such as *GitLab*, *GitHub* or *Gitea*. These allow for the granting access right to repositories, for the handling of merge requests from contributors, and for the tracking of bugs and issues. For this project the web front end Gitea was used.

For a detail description of Git it is referred to [116, 26].

4.5 Test and Verification

Every human makes mistakes. So, as long as software code is written by a person, programs will contain faults. That is already caused by the limited human capability to express a thought. Everyone, who ever was in the situation to explain a complicated subject knows how difficult it can be to put certain facts into words. This prob-

lem becomes even worse, if a thought or technical problem must be expressed in a foreign language or a computer language [29, 121]. Other sources of human error are quite evident, such as lack of knowledge, a misunderstanding of the technical problem, or simply poor concentration.

4.5.1 Terminology and Fundamental Aspects of Software Testing

Testing aims for the identification of faults in the software, and for the verification that the system behaves as specified in a given situation. In the following, a series of terms and definitions are introduced, which are commonly used in the field.

4.5.1.1 Debugging, Testing, Verification and Validation

If a test has shown that a software is erroneous, debugging usually helps to identify the part in the code, where the error occurred. Almost every modern software language provides debuggers, which enable the programmer to manually step through the routine of a program. Thus, it is more a technical feature to examine and repair the software code, rather than a means of testing it.

Testing is basically an experiment. At the beginning of the experiment a hypothesis is made about how a tested specimen will behave under certain boundary conditions. During the test, the technical system, the software, the method etc. is exposed to these boundary conditions and the resulting state of the item is checked against expectations. Like any other scientific experiment, the outcome is only valid, as long as the results of the test are repeatable ([110], p. 23). In the context of this thesis, the usage of the term *test* always refers to *functional testing*. Functional testing aims for the verification of product functionalities, that evolve during the product development and the product life cycle. It has been proposed in 1981 by William E. Howden to complement the usual *branch test-*

ing and *mutation testing* approaches. Branch testing, also referred to as *coverage analysis*, is a means of measuring to what degree a software has been tested, where mutation testing evaluates the effect that small program increments (mutations) have on the generated results [68, 19]. Further testing methodologies exist, which all have their justification within the software development process, such as *data-driven testing*, *model-based testing*, or *back-to-back testing* [121]. Yet these approaches shall not be discussed further here.

Software testing is part of the software verification process. The European Cooperation for Space Standardization (ECSS) defines verification as a

“ [...] process to confirm that adequate specifications and inputs exist for any activity, and that the outputs of the activities are correct and consistent with the specifications and input” ([44], p. 16).

While software verification simply aims for the approval that all hard requirements are met as specified, validation intends to proof that the software also behaves as expected by the user in a working environment. For a successful validation also weak criteria are important. A software which is verified with respect to specification still can be invalid for a user, if certain weak criteria are not satisfied, such as efficiency, ergonomics, or user friendliness ([121], p. 13).

4.5.1.2 Failure, Error and Fault

In his paper about *Dependable Computing and Fault Tolerance*, Jean-Claude Laprie introduces a specific terminology to describe what impairs the dependability of software: the *failure*, the *error*, and the *fault* [85].

Laprie calls a failure the occurrence of an unspecified system behavior. Quality of a failure is that a certain feature cannot be provided anymore, with the consequence that the software can become worthless for the user. A failure is triggered by an error, which

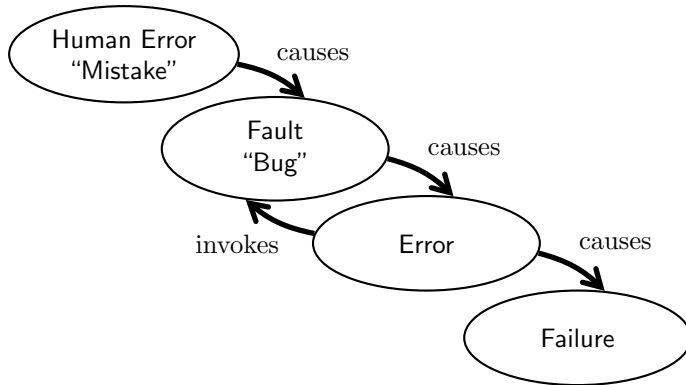


Figure 4.12: Fault, Error and Failure Chain, following [85]

Laprie basically describes as a special kind of system state. However, an erroneous software does not necessarily lead to a system failure, especially if the software provides internal error handling mechanisms.³ Errors in turn are the result of software faults, commonly known as *bugs*. Not every bug in the software does lead to an error, since the emergence of a bug usually depends on the system state. Other bugs may not become evident, simply because they are never called. According to Laprie’s terminology, bugs in turn can be invoked by errors, which are caused by other bugs, and so on. So a software might run through a series of faults and errors before a failure shows up eventually [85].

The terminology is illustrated by figure 4.12. The figure also lists the *mistake* as a human, special kind of error, the initial cause of a bug.

According to figure 4.12, any system failure can be traced back to a bug, which is initially caused by human error. The usual suspect for the origin of a bug is the programmer, who obviously has made a mistake and wrote a poor line of code, causing the error and the software to fail. But unfortunately, (or luckily for the programmer),

³A prominent example are exception throwing and catching mechanisms such as the `try`, `catch` & `throw` calls in C++ or Java applications.

it is not that easy, because bugs can emerge during the system design already and thus long before the actual implementation.

So it makes sense to improve the entire development process in a way that design flaws are detected prior to the implementation. First steps in that direction were made by leading software companies in the 1970s. The effort was driven by the finding that costs for fault removal are the higher, the later they are found ([126], p. 33).

In the scope of this project, an extensive documentation and a detailed design have been considered key with regard to a good software. Consequently, every element of the envisaged system has been documented carefully and reviewed prior to its implementation.

4.5.1.3 Testing of Object-Oriented Software

Yet, not the most sophisticated design evaluation and not even the most intense testing can guarantee that all bugs in a software are found. In their book about the testing of object-oriented software, H. M. Sneed and M. Winter compare the detection of faults with the literal search of the needle in the hay stack ([121], p. 4).

Fortunately, since the first serious efforts of establishing testing as an engineering discipline in the early 1970s, numerous methodologies have been developed in order to improve the verification process and the design of dependable software. They cover definitions, norms and standards, process models, quality assurance mechanisms, test automation, and more.

The understanding of the effect that system design or programming paradigms such as object-orientation have on the testing has evolved too. After the introduction of the first object-oriented languages in the 1980s, a popular belief was that object-orientation would not affect the testing of a software, or would even simplify it ([121], p. 22). Another common opinion was that a well-tested and well-used class can be reused easily without further testing. These assumptions turned out to be wrong [102].

In fact, certain qualities of object-oriented or reusable software complicate the verification process. Sneed and Winter mention the following four:

1. The usage of different classes in a modular system causes dependencies between the classes and the objects of these classes. A certain system functionality for example is usually not implemented by a single procedure, but a number of classes. Each class then implements the functionality of a specific domain in the process. Dependencies occur because of mutual method calls, inheritance, or the acquisition of foreign methods and attributes.
2. Classes can provide a large number of methods. The output of a method is not just a function of the method parameters, but also the state of the object, which is constituted by the class attributes. By calling a method that alters the state of the object, the output of another method can be influenced. Hence, all methods are functionally coupled by the attributes of the class.
3. The purpose a class method shall serve eventually is sometimes not evident when a class is developed. That uncertainty can result in an unpredictable number of object states, which can not all be tested.
4. The number of possible states of an object is a function of the class attributes. The more attributes a class has, the more states the object can be in. Since the number of states rises exponentially with the number of attributes, a hundred percent test of a class is simply not possible. ([121], p. 22 f.)

Similar problems apply to distributed systems.

4.5.1.4 Test Methodology

Test projects are usually organized in parallel to the actual development project. That is because the outcome of every stage in the product development process, demands for a series of specific tests. These tests must be planned, designed and conducted individually, which requires significant efforts and personnel capacities. In a classical top-down approach, it begins with the definition of acceptance tests, based on the user requirements, and ends at the bottom with code reviews, method or class tests, usually organized and conducted by the programmer.

Methodologies for the implementation of a software test can be derived from standards such as IEEE-12207, IEEE-829 or ECSS-E-ST-10-03C [71, 70, 69, 43]. These standards also define how a test must be documented. Although Institute of Electrical and Electronics Engineers (IEEE) and European Cooperation for Space Standardization (ECSS) make slightly different specifications regarding the extend of the test documentation⁴, the scope is basically the same. So, every test must pass the following phases:

1. Planning
2. Specification
3. Procedure Generation
4. Implementation
5. Execution
6. Report

Every test begins with the planning, a definition of goals and the allocation of testing resources. The phase is followed by the design and the specification of the test. For the preparation of the test

⁴E.g., ECSS further defines reviews prior and after the test activities ([43], p. 29 f.)

specification detail knowledge about the software is necessary. This is needed to estimate the behavior of the tested item in a given situation. For a distributed system, the outcome of the test specification can mean hundreds of test cases ([121], p. 44 f.).

With a certain amount of test cases, testing cannot be efficient anymore without automation. This covers server systems and tool chains for automatic software delivery, as well as scripting tools and languages for the test procedure implementation. A quite popular language for the job is the scripting language Python. In space industry Python has to compete with a vast number of scripting languages and commercial automation systems. However, some institutions have begun to establish Python as a standard and open alternative for a widely use in the testing domain [56, 18].

Outcome of the test is a test report eventually. According to the ECSS, the final test report “describes test execution, results and conclusions in the light of the test requirements. It contains the test description and the test results including the as-run test procedures, the considerations and conclusions with particular emphasis on the close-out of the relevant verification requirements including any deviation.” ([43], p. 33)

4.5.2 Test Items and Verification Process

During the previous sections of this chapter, a series of methodologies has been introduced, which have been applied on the way from the beginning of project organization until the preparation of software tests. That covered the specification of requirements based on user specifications, the domain analysis, the design of the software architecture and finally the definition of tests for the different implementation stages. This top-down approach is visualized by the V-Model in figure 4.13.

With the preliminary work being done, usually the first lines of code can be written and the first classes and methods are imple-

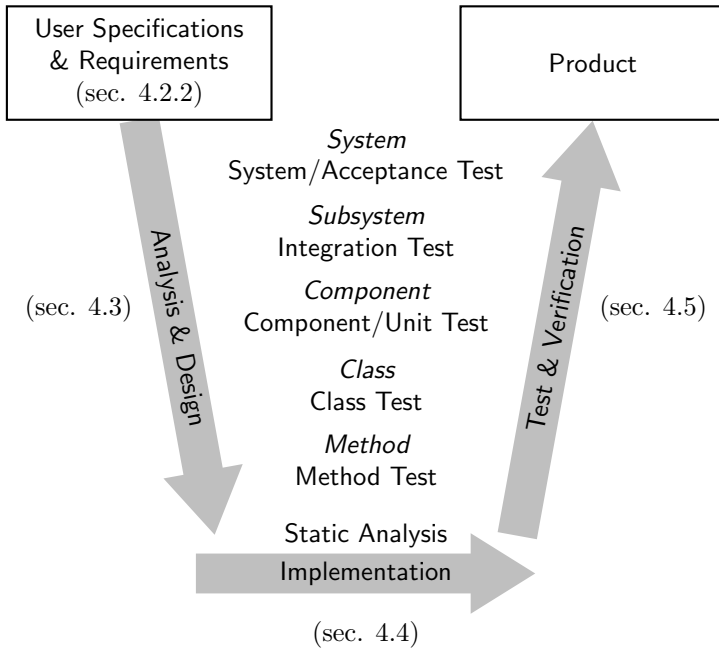


Figure 4.13: V-Model and Test Items, following [13] and [121]

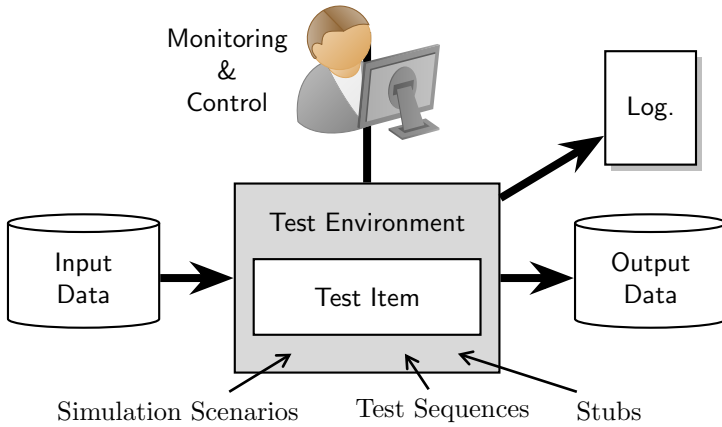


Figure 4.14: Elements of a Functional Testing System, following [121]

mented. This is when the first tests of the delivered software are conducted too.

While system analysis and design follow a top-down approach, testing and verification goes exactly the other way round. It begins with the verification of smaller items and ends with the acceptance test for the assembled system. In between, various tests are conducted depending on the tested item and the implementation stage.

A general setup for a functional testing system is shown in figure 4.14. The tested item, the specimen, is embedded in a test environment. The comprehensiveness of that environment grows with the complexity of the item. Smaller specimens of the software product are directly tested within the development framework which is used for the implementation too, while larger integration or system tests demand for more sophisticated setups.

Yet, the general purpose of each test environment is always the same. The environment must ensure that a test case is executed as specified. Therefore, it launches the defined test sequences, which execute the item and expose it to input data. Software elements, which are too complex for the test or simply out of scope, are replaced by so called *stubs*, if necessary. Stubs are simple software

blocks that can be used for data injection, the collection of data or simply as a substitution for a missing communication partner.

Complex data, which cannot be provided by simple stubs must be generated by a simulator. The worth of a simulator as a data source, especially for integration and system level tests, will be discussed in section 4.5.3.

The output data is collected in a data sink for further processing and evaluation of the test. A test environment must further provide means of logging and report generation so that the execution of the test, the coverage, and the results can be evaluated afterwards [121].

The various items and the characteristics of their test are briefly discussed in the following.

4.5.2.1 Static Analysis

“The earlier we find bugs, the easier it is to fix them. Ideally, we would like to catch errors when we make them, or as close afterwards as possible, and not ipso facto with reviewing or testing.” [88]

A good practice of finding bugs are static code analyses, sometimes referred to as static tests, although they are no test in the classical sense, as described above. For this kind of analysis, the software code is not executed (dynamically), but reviewed and checked for mistakes. These checks are conducted by the developer already and/or by means of code checkers.

Most Integrated Development Environments (IDE) for static coding languages like C/C++, provide basic code analysis capabilities already. So the developer is informed about a mistake the moment a line of code is written. After that, the code is checked by the compiler. Yet, both just provide rudimentary code checking, as they only seek for obvious mistakes, such as syntax errors, unused variables, missing initialization etc. They don't eliminate the risk of making a mistake causing a runtime error, like invalid pointers, or logical mistakes. Modern coding languages such as Rust claim to

close that gap by providing sophisticated code checking mechanisms and conclusive warning messages [58].

The need and the value of static code analysis due to the discrepancy between compiler warnings and actual flaws in the program code was recognized in the 1970s already. That time, the first code checking algorithms were introduced taking advantage of the fact that human errors during the coding process usually follows certain patterns. These patterns are recognized by code checkers [88, 75].

4.5.2.2 Method Test, Class Test and Unit Test

General purpose of a software system is to solve a problem, by creating a model of the problem domain [27]. If the paradigm of object-orientation is applied, a software unit is composed of classes, where a class implements a concept, an idea or an entity within a program [124]. A class is defined by its attributes and its functionality implemented by its methods. Methods, classes, and the software units they form must be tested successively by the developer, prior to handing the product over to the testing team.

The smallest testable item within an object-oriented software is the method. Methods are tested as function of their inputs and their parameters ([121], p. 28). The smaller and the more simple methods are kept, the more unlikely are faults and the easier they can be tested. By dividing a complex algorithm into more simple functions, the pure number of methods to be tested rises though.

The next level are class and unit tests. The challenges of testing these have been discussed in section 4.5.1 already. In conclusion, class tests are complicated mostly because of the infinite number of objects states, the dependence on the owned objects, or inheritance from foreign classes. Consequently, class and unit tests cannot be realized by means of simple input-output tests. Instead, they demand for sophisticated class drivers within the test environment, which monitor and control the state of controlled item during the test.

Modern IDEs, such as the Airbus Eclipse plugin *SimTG* for modeling and simulation purposes provide internal mechanisms for class and unit tests [140]. A popular framework for integrated unit tests in java is the JUnit testing library, available for Eclipse too [60]. Within *SimTG*, JUnit tests can be executed automatically whenever the software product is compiled.

4.5.2.3 Integration Test

During integration tests, all tested items exist in form of self-contained units. These units are the components of the developed software architecture. Thus, an integration test does not focus on a single software specimen, but the verification of the compatibility of the various items under test. In other words, the test item of an integration test is not a single self-contained object, but an abstract process that emerges through the interaction of the various units.

In section 4.3 the substantial characteristics of domain software design and the conditions for software reuse were discussed. One approach in that context was to display an architecture in two dimensions: the vertical and the horizontal dimension (fig. 4.8). Consequently, the strategy for integration testing must consider these dimensions too.

In the vertical dimension, software design strives for a hierarchy of elements within the software architecture. That hierarchy shall serve a certain process, for example a process of data handling and presentation. Such architectures are usually realized by means of multiple abstract data handling layers, where each layer implements a specific protocol, e.g. according to the Open Systems Interconnection (OSI) model [74]. In a bottom-up approach, vertical integration tests shall verify that the architecture supports the data handling process in any possible situation, and that components are compliant to the implemented data abstraction model.

The understanding of horizontal integration test is a little less abstract. Horizontal integration testing shall basically verify that

multiple elements of the same level (e.g., element A and element B in figure 4.8) are functionally compliant in any given situation. That covers the pure verification of collaboration, but also the verification of failure handling capabilities, like when one component returns corrupt data, or terminates unexpectedly ([121], pp. 197–200).

4.5.2.4 System Test and Acceptance Test

System tests and finally acceptance tests aim for the verification that all system requirements are met and that the system behaves as expected by the user. Although formally different, they are handled equally in the context of this work, as both focus on the fully integrated system.

System testing covers a wide range of different types of tests. That is because the overriding goal of system testing is not just the pure verification of all system functionalities, but also to ensure that the system remains stable under poor conditions.

System testing begins with a series of basic functionality tests, verifying that the system is compliant to its specification under nominal conditions. When these tests have been passed, the more rigorous testing begins, verifying that the system can withstand boundary conditions, which vary from nominal ([93], sec. 8). In the following, some of these system tests are introduced:

Robustness Test	A robustness test tries to determine how sensitive the system reacts on erroneous input data. The system is considered robust if a wrong input or a wrong usage does not cause a system failure.
Stress Test	Stress tests expose the tested system to boundary conditions, which exceed the specified design limits (e.g., high data rates). It is basically a test of the system's internal error handling mechanisms (sec. 4.5.1.2).

Load/Stability Test	Load and stability tests ensure that the system remains stable for a long period, when the system is working at its maximum capacity.
Reliability Test	The system reliability is quantified by measuring the time period during which the system remains stable, without the need of rework or repair.
Regression Test	Regression tests verify that the system still meets its specifications after the software has been changed or a new version has been released [93].

4.5.3 System Simulation

In section 4.5.2 it has been mentioned that testing requires input data. Simple method or class tests can be realized by means of static data sets or primitive stubs, which generate the input data during runtime of the test. Yet, the more comprehensive the tests become, the more problematic becomes the generation of valid input data, as becomes the processing of the output of the system under test.

The major inputs and output of an MMOS are the TM/TC frames exchanged with the satellite. Since system testing cannot be done with a real satellite in orbit, simulating the spacecraft functionality is the general practice [47]. Figure 4.15 depicts the target setup with a number of simulated satellites connected to the developed system under test.

For this work an industrial simulation IDE called *Simulation Third Generation* (SimTG) was used. SimTG is an *eclipse* plugin, allowing for simulator implementation as well as for simulator execution [140].

A simplified architecture of a simulation environment like SimTG is shown in figure 4.16. Such environments consist of three basic

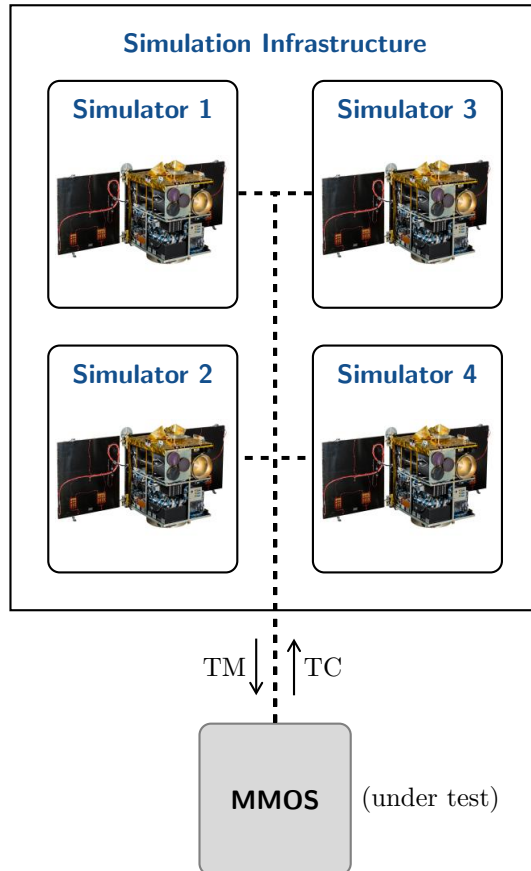


Figure 4.15: Simulated Constellation - Verifying an MMOS requires the simulation of multiple satellite systems simultaneously.

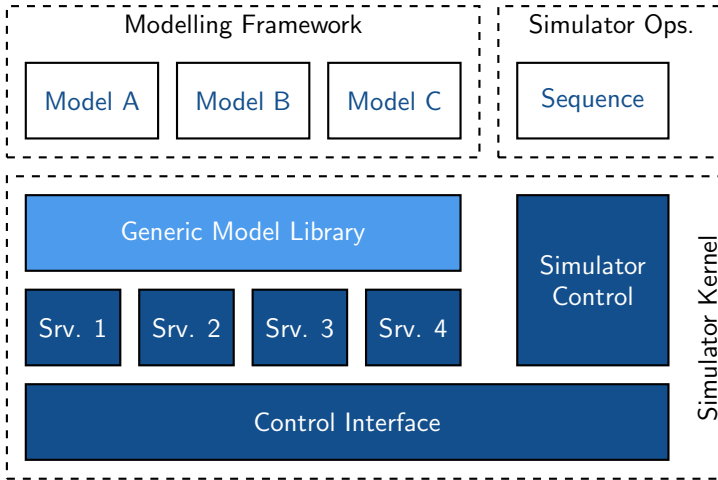


Figure 4.16: Simplified Simulator Architecture, following [22]

elements. The *modelling framework* is the element which allows for the implementation of simulation models. These models usually represent a system, a device, a physical phenomenon, or what ever shall be simulated. During runtime these models are executed by the *kernel*, which can be monitored and controlled from the *simulator operations tool*.

4.5.3.1 Modelling Framework

Models within in a system simulator as shown in figure 4.16 are discrete-time models, which means that they map the discrete state of a system over time. In control theory it is general practice to represent a system by means of discrete-time models, whenever a controller for that system is designed ([1], p. 268).

The simulation models, which define the scope of the simulation are implemented within the modelling framework (*SimMF*). Models, created within SimTG are SMP2 compliant, which supports model reuse from different simulator projects [38].

SimTG uses the programming language C++ for model implementation. In the first instance, a simulation model is nothing more but a class. SimMF supports the user in the specification of these classes, which covers the declaration of attributes, the declaration of member functions, the declaration of interfaces, and the declaration of dependencies (e.g., to other simulation models). Furthermore, SimMF supports the user in the specification of those model characteristics, which go beyond a traditional UML compliant declaration of a class. This covers for example the definition of initialization values, variable ranges, or advanced visibility settings. The model code is automatically generated according to these specifications.

After model creation, the developer is in charge of implementing the model functionality. Three different member function types must be implemented.

Executable Functions

During runtime, simulator models are executed by the simulator kernel. Executable member functions refer to all those methods which can be called from the kernel. The value of this is manifold. Basically, the entire *simulator scheduling* as will be described in section 4.5.3.5 relies on this functionality. Apart from that, calling executable functions allows for the implementation of model self-tests, for failure injection, or for a direct manipulation of the model state.

Model Function: *init()*

init() is an executable function that is called once during a default simulator initialization process that stands at the beginning of each simulation run.

After a model has been defined, SimMF automatically generates the program code including all class constructors. The *init()* method

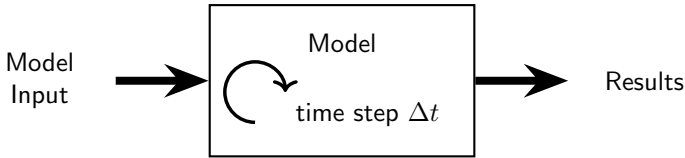


Figure 4.17: Simulator Model Working Principle - Model inputs are processed during a time step, and propagated afterwards.

is used to implement all model initialization steps, which are not covered by the automatically implemented model constructors.

Model Function: *step()*

step() is the executable function that actually defines the behavior of the model. As such, it is the function that implements the numerical algorithms that represent a certain physical phenomenon, or quality of a simulated component.

Unlike *init()*, *step()* is not called just once, but cyclically with a constant period Δt . At the beginning of each step stands an update of all model inputs. Following that, a time step is executed. The scope of a time step can be manifold. It can be an integration step, a run through a state machine, or any sort of algorithm representing a functional behavior. After that, the model outputs are propagated, and can be used by other models. The process is illustrated in figure 4.17.

4.5.3.2 Kernel

The simulator kernel is responsible for model execution. It provides a series of mechanisms (services) that support the simulation, such as time services providing the simulation time to models, scheduling services actually executing those models, logging services etc.

Industrial simulation frameworks like SimTG [140], already provide a formally verified kernel. Thus, a simulator does not need to be developed from scratch whenever a new system simulation needs

to set-up. Instead, simulator development can focus on modelling, integration and testing, and scenario generation eventually.

Once all models have been completely implemented, they are compiled together with the kernel library. The result is a software binary that can be executed and controlled from the simulator operations tool.

4.5.3.3 Simulator Operations Tool

Within SimTG, *SimOPS* refers to the tool that takes care of simulator monitoring and control. It executes and commands the compiled simulator kernel through an API. That kernel API allows for the call of various kernel functions from simulator sequences to be created by the user. SimOPS is based on the java coding language for the realization of these sequences. They define which models are to be executed during a simulator run, the duration of that run, as well as the simulation boundary conditions (epoch, initial model stats, etc.). A generic sequence contains the following steps:

1. Creation of a simulator process (Does *not* mean starting the simulation!)
2. Creation of model instances
3. Setting of the frequencies of the models' *step()* function calls
4. Connection of the model interfaces
5. Simulator default initialization
6. Override of the default initialization
7. Start of the simulation
8. During the simulation runtime, sequences can be used to perform model tests, or to monitor & control the simulation. If none of this is required, the sequence just idles for as long as the simulator is supposed to run.

9. Stop of the simulation
10. Termination of the simulator process.

Beyond this basic functionality, SimOPS provides a series of additional features, including but not limited to: a model debugger, a failure injection, logging, data sampling, and a user interface.

4.5.3.4 Modeling

This thesis follows the model concept as defined by H. Stachowiak in 1973. In his book about General Model Theory, Stachowiak defines a model as an

“image [...], a representation of a certain (real) original [...].” ([123], p. 129)

According to this definition, a model is characterized by three basic features:

1. “A model is an image/representation of *something* natural or artificial.”
2. “A model does not capture *all* attributes of the original, but only those which are relevant for the model creator and/or the model user.”
3. A model may provide functionality, that “cannot be assigned clearly to an original (functionality). (Instead), models provide substitute functionality”, which depend by whom, when, and why a model is used. ([123], p. 131 ff.)

The following example shall illustrate the concept (fig. 4.18). Figure 4.18a shows an image of a model locomotive. First, it is the miniaturized version of the original German BR 216 series. Second, a model train of course cannot provide all features of the original. The model has no brakes, no air pressure system, no heating, etc.



(a) Model Locomotive



(b) Electric Motor



(c) Wheel Set



(d) Coupling

Figure 4.18: Model of a BR 216 Diesel Locomotive

Third, certain features of the original are realized by substitute models. So, instead of a diesel engine, the model has an electric motor, but the observable functionality: the train moving forward, stays the same. Other features like the wheel set and the coupling are very simplified and hardly resemble the real implementation.

For the realization of a satellite system simulation, the following items need to be modelled:

- satellite components (devices, structure, ...)
- interfaces (power lines, data links, ...)
- orbital dynamics
- space environment (atmosphere, magnetic field, ...)

Prior to the modelling process, the relevant physical properties of the originals need to be determined. Of interest in space system simulation are usually the electrical properties, the mechanical properties and the thermal properties, which is why a component model is normally decomposed into the appropriate sub-models (fig. 4.19).

Not every property of a component needs to be simulated though, instead the scope of a model should be kept to a minimum and only the relevant properties should be simulated. The identification of the relevant and non-relevant sub-models is part of the model design process. For example, the simulation of a reaction wheel for the verification of an attitude controller does not require the simulation of the component's thermal behavior.

A model decomposition into separate, closed sub-models is not trivial though, because different physical qualities can be functions of each other. For instance, the thermal condition of a device (e.g., a solar cell) normally affects its electrical characteristics.

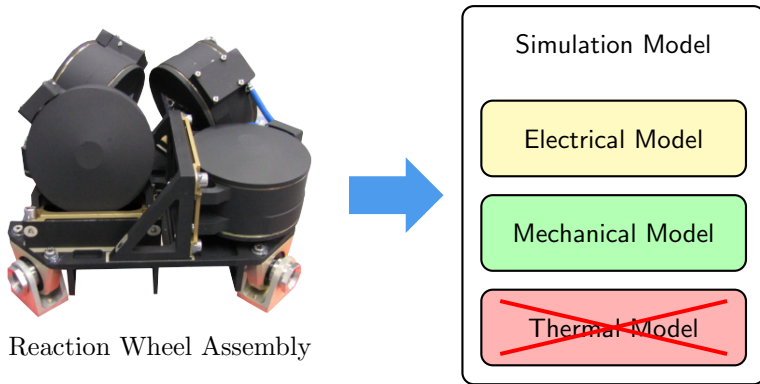


Figure 4.19: Properties of a Satellite System Model - The selection of relevant properties depends on the individual use case. E.g., for the simulation of a reaction wheel, thermal properties are usually not relevant.

4.5.3.5 Simulator Model Scheduling

Within a simulator architecture as sketched in figure 4.16, the term *scheduling* refers to the frequency of the model step function calls by the kernel. Before discussing aspects of the model scheduling, some time definitions as used in satellite system simulation shall be introduced.

Simulator Session Time (SST) - “Zulu” The SST is the *actual* time shown by the clock on the wall. Within satellite operation and satellite simulation environments, it is usually indicated in UTC.

Simulation Run-time (SRT) The SRT is the simulated time. All time dependant processes within the models are in reference to the SRT. It starts at 0 s and counting. If the simulation runs at real time speed, SRT and Zulu-Time are synchronized. If the simulator execution is paused, SRT counting pauses too.

Simulated Mission Time (SMT) The SMT can be any time in the past, the present and the future, depending when the simulated scenario takes place. The SMT is usually, set actively to a specific Modified Julian Date (MJD) at the beginning of the simulation. The SMT is always in sync with the SRT ([47], p. 156).

After the models have been implemented, they are executed as specified by the simulation sequence (sec. 4.5.3.3).

A simulation sequence specifies which model and how many instances of each model shall be executed. Furthermore, a simulation sequence allocates a scheduling frequency and an initial offset to each model instance. The scheduling frequency determines the period Δt_{SRT} between two calls of the model step function, where the offset specifies the time (in SRT) of the very first call. An example scheduling scheme is shown in figure 4.20.

Depending on the size and the complexity of the simulation, the scheduling frequencies can have a huge impact on the fidelity and the real time capability of the simulator. The right period is a parameter that must be found for each model individually. It is the result from a trade-off between model accuracy and simulator performance. If the frequency is too low, the numerical simulations could be insufficient. If the frequency is too high, the entire simulator could lose its real-time capability. The latter is the case, when the time for the execution of the model step functions comes close to, or even takes longer than the specified scheduling periods. In such a case a rework of the simulator scheduling in favor of a lower frequent model execution is necessary.

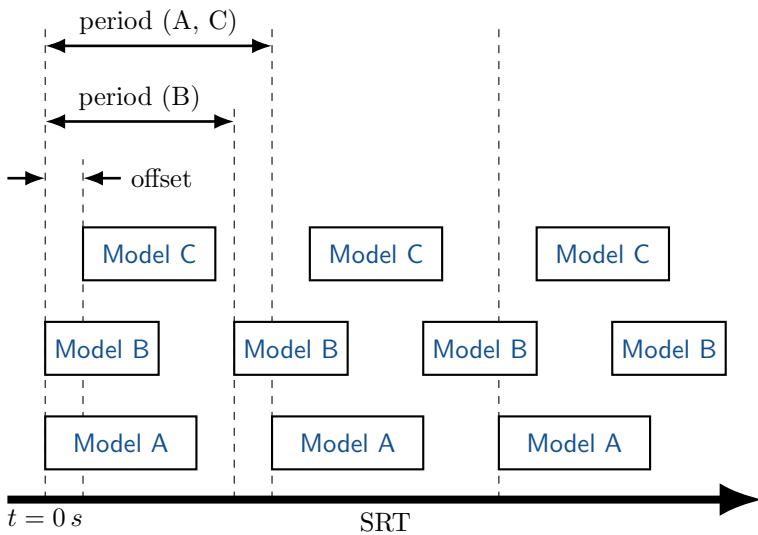


Figure 4.20: Example Scheduling Scheme, following [47] - Scheduling periods and the scheduling offset of three simulation models over SRT. The model execution times (Zulu) are indicated by the bar widths. Qualitative display only.

MMOS Domain Engineering

In a top-down approach, chapter 4 introduced those methodologies, which were applied in the scope of this project.

Following that approach, this chapter deals with the domain engineering process of the Multi-Mission Operations System (MMOS). It covers the discussion of the system quality requirements, an introduction of the MMOS subsystems in the scope of a domain analysis, and ends with a brief discussion of the outcome system design.

Chapters 6 and 7 will then go further into details of the Mission Planning System (MPS) design, and aspects of system verification.

5.1 Quality Requirements

Each application needs to be designed according to user requirements. An Operating System (OS) like the MMOS shall be useful for a wide range of people, such as: satellite controllers, flight directors, mission planners, payload scientists, engineers, etc., where each group has different expectations on the system, in terms of functionality *and* in terms of quality.

In contrast to functional requirements (appendix. B.1), which define *what* must be implemented, non-functional requirements spec-

Table 5.1: Mutual impairment of the five most significant non-functional requirements and the effect of an extended system functionality to the fulfillment of these criteria, following [8]

	<i>Efficiency</i>	<i>Reliability</i>	<i>Usability</i>	<i>Security</i>	<i>Maintainability</i>
Efficiency	█	-	0	-	-
Reliability	-	█	+	+	+
Usability	0	+	█	-	0
Security	-	+	-	█	+
Maintainability	-	+	0	+	█
Functionality	-	-	0	-	-

+ positive influence, - negative influence, 0 no significant influence

ify *how* the system shall be designed. Non-functional requirements¹ apply to the entire system. Yet, a clear distinction between both types is not always so easy, since technical requirements can have an effect on the required functionality too ([8], p. 109). E.g., *security* is a common non-functional requirement for a software system. For the implementation of an encryption methodology however, it can become a functional requirement.

Numerous non-functional requirements are cited and used in literature to describe the quality of a technical solution. A study by [89] has identified more than hundred². Yet, the meaning and the dissemination of the various quality criteria strongly depends on the domain and the kind of application.

¹sometimes called technical requirements, or quality requirements

²That list however should not be taken for granted, as a lot of mentioned criteria are derivatives of each other, e.g. *reliability*, *verifiability* and *testability*. A reliable software of course must be verifiable. And a verifiable software of course must be testable too. Also the work lists a lot of synonymous requirements, e.g. *expandability*, *extendability* and *extensibility*.

The five, most significant quality criteria are listed in table 5.1. They are *efficiency*, *reliability*, *usability*, *security*, and *maintainability* [8]. A general problem with quality criteria is their contradictory nature, which makes it impossible for an architecture to cover all of them equally. E.g., an enhanced efficiency generally impairs the reliability of a system. The effect becomes worse the more functionality the architecture provides. Finding a solution in between is a well-known problem in software engineering.

Consequently, a prioritization is necessary, and a focus on those quality criteria, which have the biggest value for the user of the software. The following sections introduce the considered technical requirements for the MMOS. According to [8], the discussion distinguishes between *runtime requirements* and *non-runtime requirements*.

5.1.1 Runtime Requirements

Runtime requirements are those quality criteria, that take effect during execution of the software. A formal verification of such can be difficult, because the degree to which quality aspects have been fulfilled can only be assessed when the software is actually used ([8], p. 114). Hence, the evaluation of these quality issues is the matter of a software validation and not of the prior formal verification.

5.1.1.1 Efficiency and Automation

It is assumed that the key to efficient constellation operations is automation. Within the MMOS, basically everything shall automated that is too time consuming, generates too much workload, or is prone to human error.

In the area of satellite operations, it is common practice to refer to an OS as automatic the moment it is capable of performing single tasks autonomously, i.e. the generation of a command stack from plan. But that is not the full story. So, what makes a fully automatic

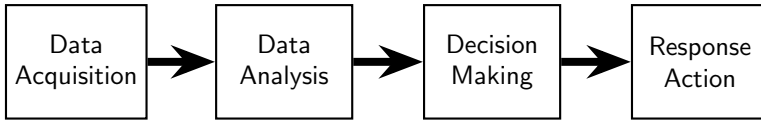


Figure 5.1: Four-stage Model of Information Processing, following [99]

system then? For that to understand, it is necessary to know what automation means.

First of all, automation means that a task formerly done by a human is fully or partly carried out by a computer. Whenever that happens, a decision must be made to what degree automation shall be implemented. In other words: How much of the task shall be done by the computer, and how much shall the human still be involved? This requires an analysis of how human and computer interact, by whom an action is done, and how that action is communicated. In that context, Sheridan and Verplank defined the *10 Levels of Automation in Man-Computer Decision-Making* ([118], p. 8-17 ff.), which specify ten grades between a task completely done by a human and a fully automatic implementation (tab. 5.2).

Secondly, within a mission OS, automation is the implementation of a process; a process of monitoring and control of a satellite mission. A principle four-stage model of that control process is shown in figure 5.1.

A control action is always the result of a former sensor data acquisition, an analysis of that data, a decision making, and an implementation of that action. Parasuraman and Sheridan found that each of these four stages can be automated at different levels³. Whenever an automatic system shall be implemented, the selection of the respective automation level depends on a series of boundary conditions, such as: the workload, the costs, the criticality of the

³Parasuraman and Sheridan introduce their findings with the words: “Consider the following design problem. A human operator of a complex system provided with a large number of dynamic information sources must reach a decision relevant to achieving a system goal efficiently and safely.” [99] In a way, this shows how valuable their approach is for this work.

Table 5.2: Levels of Automation in Man-Computer Decision-Making - for a single elemental decisive step [118]

Level	Description
1	Human does the whole job up to the point of turning it over to the computer to implement.
2	Computer helps by determining the options.
3	Computer helps determine options and suggests one, which human need not follow.
4	Computer selects action and human may or may not do it.
5	Computer selects action and implements it if human approves.
6	Computer selects action and informs human in plenty of time to stop it.
7	Computer does whole job and necessarily tells human what it did.
8	Computer does whole job and tells human what it did only if human explicitly asks.
9	Computer does whole job and tells human what it did, if the computer decides he should be told.
10	Computer does whole job, if it decides it should be done, and if so tells human, if it decides he should be told.

action, the reliability of an implementation, the presentation of data etc. Considering these and other boundary conditions upper and lower bounds can be determined for each stage independently [99].

However, a poor automation can also impair the system quality, and therefore increase an operator's cognitive workload. This is the case when a too rigid form of automation retains valuable information, so that the operator is not able to trace a decision process, or the cause of a failure anymore [99]. Even a balanced level of automation can impair the system quality. That is the case, when "the burdens associated with managing automation [...] outweigh the potential benefits" [80], for example when the automation is "clumsy [, and] difficult to initiate and engage" [99].

So, the one question that needs to be answered is: Which M-MOS system functionality (stage in the control process) shall be automated to what extend? Answering that is actually quite the trade-off and has been the topic of research since computers have started taking over human tasks.

Table 5.3: Simplified Level of Automation Taxonomy (LOAT) Matrix, as defined by Save and Feuerberg [113], based on the work of Parasuraman and Sheridan [99] - table from [112]

A	B	C	D
Information Acquisition	Information Analysis	Decision and Action Selection	Action Implementation
\downarrow <i>Level of Automation</i> \longrightarrow <i>Process</i>			
A0	B0	C0	D0
Manual Information Acquisition	Working Memory Based Information Analysis	Human Decision Making	Manual Action and Control
A1	B1	C1	D1
Artefact-Supported Information Acquisition	Artefact-Supported Information Analysis	Artefact-Supported Decision Making	Artefact-Supported Action Implementation
A2	B2	C2	D2
Low-Level Automation Support of Information Acquisition	Low-Level Automation Support of Information Analysis	Automated Decision Support	Step-by-Step Action Support
A3	B3	C3	D3
Medium-Level Automation Support of Information Acquisition	Medium-Level Automation Support of Information Analysis	Rigid Automated Decision Support	Low-Level Support of Action Sequence Execution

Table continues on next page.

Table 5.3: Simplified LOAT Matrix, as defined by Save and Feuerberg [113], based on the work of Parasuraman and Sheridan [99] - table from [112]

Information Acquisition	Information Analysis	Decision and Action Selection	Action Implementation
A4	B4	C4	D4
High-Level Automation Support of Information Acquisition	High-Level Automation Support of Information Analysis	Low-Level Automatic Decision Making	High-Level Support of Action Sequence Execution
A5	B5	C5	D5
Full Automation Support of Information Acquisition	Full Automation Support of Information Analysis	High-Level Automatic Decision Making	Low-Level Automation of Action Sequence Execution
		C6	D6
		Full Automatic Decision Making	Medium-Level Automation of Action Sequence Execution
			D7
			High-Level Automation of Action Sequence Execution
			D8
			Full Automation of Action Sequence Execution

To support this trade-off, a taxonomy has been applied that subdivides each stage in the control process into specific automation levels. That taxonomy is called the LOAT matrix [113]. In a con-

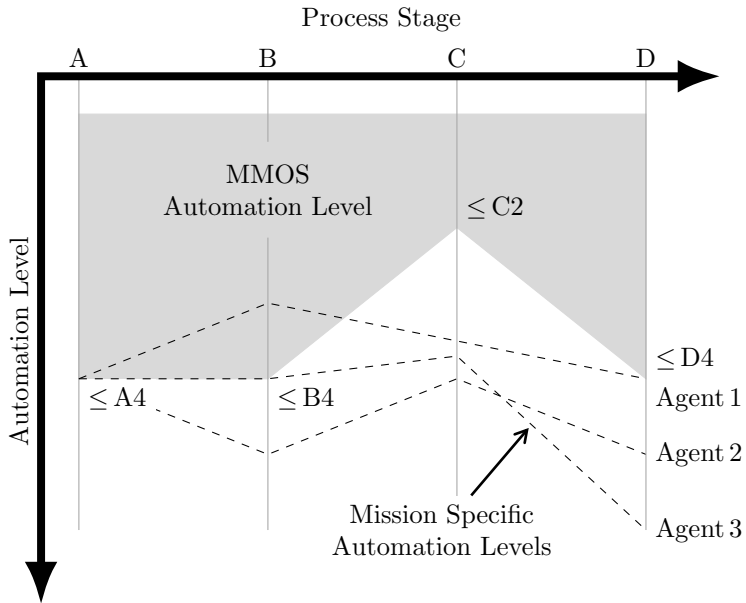


Figure 5.2: Foreseen MMOS Automation Levels according to the LOAT - Just a qualitative figure. Individual automation levels need to be adapted according to the respective system functionality. Mission specific decision making as well as action implementation can be realized by individual agents.

densed form it is shown in table 5.3. The full matrix, including detail level descriptions, is available in appendix A.2. In this taxonomy, A, B, C and D refer to the control stages in figure 5.1, where for each stage specific levels of automation are defined. The higher the number, the less involved is the human into the respective stage.

The striven levels of automation within the MMOS control process are indicated by the gray area in figure 5.2. The indicated bounds (A4, B4, C2, D4) are according to the LOAT matrix.

The acquisition of data from the operated systems shall be fully automatic (A4). This covers the reception of TM packets, the extraction of parameter values from these packets, and the writing of that data into a persistent database. The definition of the received

data types is a configuration made by the user though, as well as the definition of parameter limits, and the type of display.

Some of the received data must be further processed. E.g., the received satellite position and velocity information must be integrated for orbit prediction and the calculation of pass times. These computations are done automatically too, based on pre-defined user specifications. Thus, the upper bound for data analysis B4.

The decision making based on the analysed data is highly mission specific. Consequently, the MMOS can only provide a very limited automation level during this stage (C2). The MMOS shall only *support* a decision making process, e.g. by evaluating certain system metrics. A full/high-level decision making is the task of so called Agents, which must be designed and implemented for each mission individually. Agents are therefore no feature of the baseline OS that is the MMOS. By means of Agents various automation levels can be achieved during each stage in the control process (dashed lines in figure 5.2).

Once a decision is made, a satellite activity must be scheduled. After activity creation, either by an operator, or by an Agent, the MMOS is in charge of handling that activity. That covers all steps from the scheduling and the release of commands, until the verification of the activity execution. An operator however must be able to monitor and interrupt the process. The respective automation level according to the LOAT is D4. An automation level higher than D4 can be achieved, if the activity execution is monitored by an Agent.

It is important to know that the automation levels specified in figure 5.2 (A4, B4, C2 and D4) are just upper bounds. This means that an operator granted with respective rights must always have the opportunity to intervene the process and/or to trigger system functionality manually. Situations where this might become necessary are manifold, like the following:

1. An Agent might be erroneous and the automatically scheduled activities would jeopardise the mission.

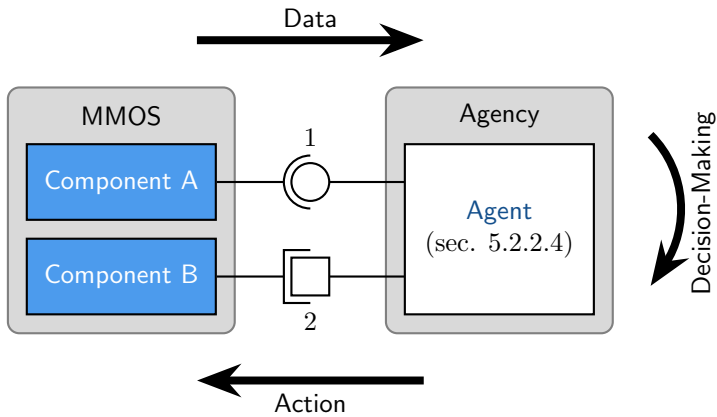


Figure 5.3: Standardized Interfaces for System Extensibility - Different interfaces are needed for the provision of acquired data to Agents (1) and for the insertion of actions implemented by these Agents (2).

2. The automatic decision making or the automatic action implementation might fall apart. In that case the operations team must be capable of scheduling satellite activities manually (e.g. by releasing manual commands).
3. Full and high-level automation can cause skill-degradation [99]. To prevent this from happening the operations team must be capable of performing low-level operations for training purposes.

5.1.1.2 Extensibility

As figure 5.2 indicates, various missions could demand a significantly higher level of automation than provided by the MMOS. The means of closing that gap are so called *Agents*, which are individual software components, tailored to their specific mission. Unlike other components, Agents are no element of the baseline MMOS. Consequently, the MMOS must actively support the addition of external software.

To fulfill that extendability requirement, the system must provide a number of well defined and documented interfaces, enabling a development team to add mission specific functionality to the MMOS. According to the process model in figure 5.1, this requires generic interfaces for the provision of acquired and/or analyzed data to those Agents, as well as interfaces enabling Agents to inject decisions into the MMOS (fig. 5.3).

These interfaces as well as the concept of Agents are introduced in section 5.2.2.4.

5.1.1.3 Scalability

Scalability is a frequently referenced non-functional requirement for computer and software systems, even though it has never been comprehensively defined [64, 89]. In 1990, after a failed attempt of defining it algebraically, one author even encouraged “the technical community to either rigorously define scalability or stop using it to describe systems”, as it would be “about as useful as calling [a system] *modern*” [64].

Yet, every distributed system, such as the MMOS, must cope with growth and an increased load eventually. In that context, the technical community (if there is any) has agreed on a certain terminology in order to describe the capability of a system to deal with these issues. Such a terminology is considered useful even though scalability as a measurable characteristic of a software system is difficult to evaluate and to express in mathematical terms.

Following A. Bondi, scalability describes

“the ability of a system to accommodate an increasing number of elements or objects, to process growing volumes of work gracefully, and/or to be susceptible to enlargement.”

Consequently, a system is *not* scalable, if

“the additional cost of coping with a given increase in traffic or size is excessive, or [the system] cannot cope at this increased level at all.” [14]

In that context “cost can be quantified in many ways, including but not limited to response time, processing overhead, [...] memory, or even money.” [14] Different types of scalability have been defined, because the performance of a system can be compromised by growth in many ways.

Load Scalability

Load Scalability describes the ability of a system to perform properly at increased loads. Measures for the quantification of load scalability can be manifold, such as response times, memory consumption, CPU utilization etc.

Space Scalability

A system is considered space scalable, if an increased number of supported items/objects does not result in an unacceptable usage of memory. Sometimes space scalability can only be achieved at the expense of load scalability, as will be discussed later on.

Space-Time Scalability

Similar to space scalability, space-time scalability requires that the system is capable of supporting an increased number of objects. Yet the measure for space-time scalability is not memory usage, but the maintenance of a sound system performance. Space-time scalability is an important requirement for search engines and databases, which must be able to return the requested data within an acceptable time period, even though the number of encompassed objects increases by an order of magnitude. Consequently, space scalability is a necessary requirement for space-time scalability.

Structural Scalability

Structural Scalability describes the general ability of a system of accommodating new objects. The number of objects a system can handle is usually limited by the size of the address range. E.g. if objects are identified by means of an 8 bit unsigned integer, the number of handled objects is limited to 256. [14]

Unlike space, space-time and structural scalability, which are architecture inherent, load scalability can be achieved by exploiting the capacities of the computing hardware. Given the availability and the increasing technical capabilities of such, load issues are gladly solved by means of parallelism, multi-processing or simply by installing more powerful hardware, even though those issues could be solved more efficiently, as the following example shall demonstrate.

E.g., the problem could be to solve the linear system $Ax = b$. If matrix A is of size $n \times n$, a Gaussian LU-decomposition to solve the equations would require approximately $1/3 n^3$ operations. Within numerical analysis, the asymptotic big O notation is generally used to quantify the computational effort of such a problem. In this case, the problem scales in the order of

$$\mathcal{O}\left(\frac{1}{3}n^3\right), \quad n \longrightarrow \infty. \quad (5.1)$$

If the Cholesky method is used instead, the same problem can be solved with only half as many operations: $\mathcal{O}(1/6 n^3)$ ([32], p. 78, 88). This example shall demonstrate that with every performance issue, at first the underlying algorithms (respectively the scheduling) should be questioned prior to the attempt of diminishing the problem by an overhead in computing power. Further options for the improvement of load scalability according to [14] are:

- identification of unproductive execution cycles
- reducing sojourn times in cycles.

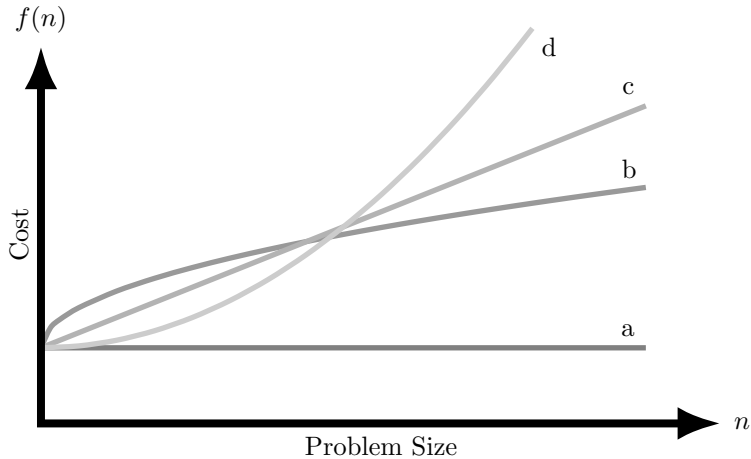


Figure 5.4: Increase in Cost/Effort as Function of Problem Size n - (a) Effort is a constant and thus now function of the problem size, (b) sublinear, (c) linear and (d) superlinear increase of effort.

- avoiding deadlock situations
- altering scheduling rules
- using options of exploiting asynchronous/parallel execution

However, considerations with respect to load-scalability require detail knowledge about the system functionality, and the implemented algorithms already. For the discussion of the architectural requirements: space, space-time, and structural scalability, it is reasonable to specify the demand in a less formal way [12].

In the following discussion the problem size n shall be the number of operated satellites. Figure 5.4 shows in which manners that number can increase the operational costs. Case a) is a rather theoretical example, where the effort of operating a distributed system is a constant and no function of the number of satellites at all.

If the satellite systems are organized within a hierarchical architecture, for example under the authority of a planning entity, the

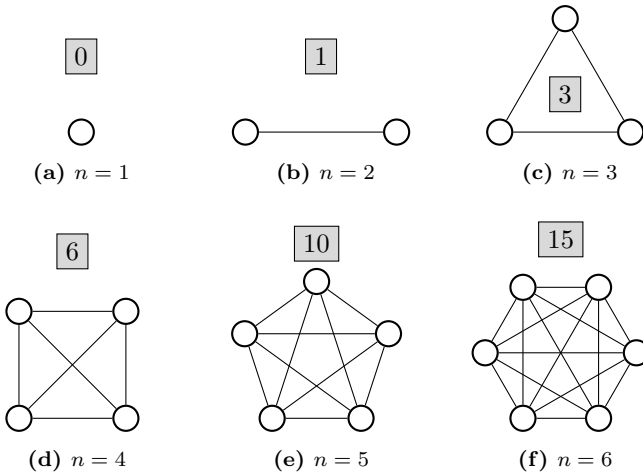


Figure 5.5: Number of connections in Peer-To-Peer Networks of n Nodes - An example of a superlinearly scaling problem, as the number of connections rises increasingly faster than the number of nodes

problem would most certainly scale sublinearly (case b). Indeed, the set-up of a central control entity requires a certain overhead in costs. Yet, the additional costs decrease with every new operated satellite.

A linear increase in costs (case c) turns out, if each satellite is operated individually. There is no overhead due to a central control entity. Yet, since the same planning capabilities need to be implemented for every single satellite again and again, the additional expenses stay the same for each new satellite.

Case d) describes the situation of operating a distributed system with an increasing complexity with respect to the number of satellites. Due to the dependencies between the satellites and the additional effort of coping with them, such a problem scales superlinearly. [12]

Eventually, the requirement for the MMOS is two fold. At first, the OS must be able of coping with distributed satellite systems that

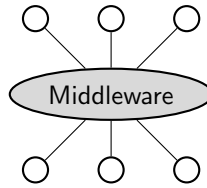


Figure 5.6: Network Topology as Selected for the MMOS - The improved space scalability, comes at the expense of load scalability, as the middleware has to handle the entire traffic.

scale as shown in figure 5.4. Secondly, the MMOS as a distributed system *itself* must be capable of growing. That is quite obvious, because with an increasing number of operated satellites the MMOS grows too, as well as the group of people using it. All of this results in an increased competition for system resources and therefore an increased number of control and scheduling processes, an increased number of accesses, an increased traffic, and an increased amount of data to be handled and stored. The demand for scalability derives from the size and the operational concept of the supported mission types, as discussed in section 3.2.

In terms of space consumption, a sublinear scaling implementation is envisaged. An example where this has not been achieved is shown in figure 5.5. In the displayed peer-to-peer networks the number of connections grows increasingly with every new node. Hence, the problem scales superlinearly. Consequently, the MMOS shall feature a star like network topology (fig. 5.6) with a central node that manages the connection between all the entities in the system. The example shows that resolving an issue in terms of scalability can impair the system performance. In a network topology as shown in figure 5.6, the entire traffic is routed via the middleware, which must be appropriately load scalable.

The balance between service quality (performance) and scalability is an individual optimization problem where either the quality of the solution must be maximized, or the costs must be minimized.

A good scaling solution most certainly comes at the cost of poor service quality, while optimal solutions usually scale superlinearly. If however the operated system size is known or definitely will not grow beyond a critical limit, the latter does not necessarily impair the system scalability. [12]

The trade-off introduced here applies to the entire system architecture, as well as to the design of units and classes, as the quality of low-level implementations directly affects the quality of the entire system.

5.1.1.4 Reliability

Reliability is a measure of the correctness of software with respect to time. It is usually quantified by the count or the percentage of operations completed correctly within a given period, or by the average duration between two software failures ([135], p. 274).

A sole demand for reliability is a quality requirement hardly to fulfill though, because the correct working of a software depends on a series of boundary conditions, like: Who is using the software? How is the quality of the input data? For how long and how often will the software be used? How large is the throughput? From the answers to these questions a series of further quality requirements can be derived. The reliability of the software is then nothing more than a manifestation from the fulfillment of these requirements.

Robustness

Robustness describes the capability of a system to handle erroneous input data without an impairment of its operability. Primarily, the MMOS has to deal with two kinds of inputs: satellite TM and user inputs. Both can be invalid in different ways. In terms of satellite TM, the system must be able of coping with bad data that can be corrupted due to transmission errors, or some kind of malfunctioning. Invalid data must be recognized as such, and presented appropriately. Similar applies to user inputs. The assumed group

of people using the MMOS consists of trained personnel. Due to the given complexity however, the risk of an incorrect usage cannot be eliminated. This is why the system must be capable of keeping quality of service upon wrong usage, as the users must be prevented from making erroneous inputs as good as possible.

Availability

Satellites that are not coasting, or about to be injected into a target orbit, are usually active around the clock. This requires the OS not only to be up and running 24/7 as well, but also constantly available. Availability is a measure to which degree a targeted operational time has been reached by the system. It is quantified by the ratio of the actual up time and the target up time (following [135], p. 267).

Stability

Stability describes the ability of a system to handle an increased data throughput, or an increased usage over a longer time period ([93], p. 194). The stability requirement is a direct consequence from the demand for system availability and the expected system load.

Consistency

If a system is lacking consistency, different entities relying on the same data can produce conflicting outputs. An area where this can become a problem is mission planning. If the mission schedule is inconsistent, or the state of the operated system is not mapped unambiguously, conflicting satellite activities could be the result. In a worst case, this could jeopardize the mission.

For that reason, the MMOS must ensure that a used information, data point, or metric is *true*, meaning that two instances referencing the same data point get the same result in any situation. This is why the MMOS shall implement a very strict single source of truth (SSOT) policy.

Verifiability & Testability

The reliability of a system of course must be verified. Consequently, a reliable system must be testable too. Unlike the other quality attributes though, verifiability and testability have no direct value for the user. In this sense, it is quite astonishing that authors like [89] list verifiability and testability as self-contained quality criteria of software.

5.1.2 Non-Runtime Requirements

Unlike runtime requirements, which take effect when the software is executed, non-runtime requirements cover the static qualities of the software design.

5.1.2.1 Reusability

A software component is reusable if it can be used in different applications. Thus, reusability “indicates the relative effort required to convert a software component for use in other applications.” ([135], p. 284)

Due to the problems and the characteristics of object-oriented software (sec. 4.5.1), this requires a certain degree of thoroughness and a comprehensive system analysis. Components can only be reused, if the software supports modularity, and the reused element is well documented and not designed for a specific application. Beyond that, the adherence of standards, a rigorous maintenance of the libraries, and permanent regression testing are necessary ([135], p. 284).

In this specific context, software elements shall be reused in two different manners.

1. In accordance with the common understanding of reuse, single software items such as classes, interfaces etc. shall be reused throughout the development project wherever possible. This

requires a maintained library of generic items, which developers can utilize for the implementation of their product.

2. Within the MMOS, reuse further means that multiple instances of one and the same component are utilized by different missions. A mission specific behavior of a reused component is achieved through configuration.

5.1.2.2 Configurability

The system configuration is quite a crucial aspect in terms of multi-mission operations. Current OS, designed for single-mission operations, are usually configured by means of a Satellite Reference Database (SRDB)⁴, which features those mission specific information the OS requires to establish a communication with the satellite. This covers for example TM, TC, and parameter definitions.

Purpose of these SRDBs is to simplify and to standardize the mission specific configuration, and to support the exchange of mission information between different OS. An example is the *SCOS-2000* standard [52], which is used for the configuration of the eponymous Mission Control System (MCS).

An effort that is often underestimated, is the part of the OS configuration, which is not mission specific. Unfortunately, it is not sufficient just to configure single components for a mission. The OS *itself* needs to be configured too.

There are a lot of variables, which are not necessarily mission specific and which therefore cannot be set by means of an SRDB. For example, the OS must be orchestrated⁵, it must be connected to a number of ground stations, and users must be registered at the system. The task of setting up an OS is usually quite time consuming, requires detail knowledge about the architecture, and

⁴In some applications the SRDB is referred to as Mission Information Base (MIB), which is also the term used in this project.

⁵In this context the term orchestration refers to the process of selecting, launching, and executing all necessary system components (sec. 5.3.3.1)

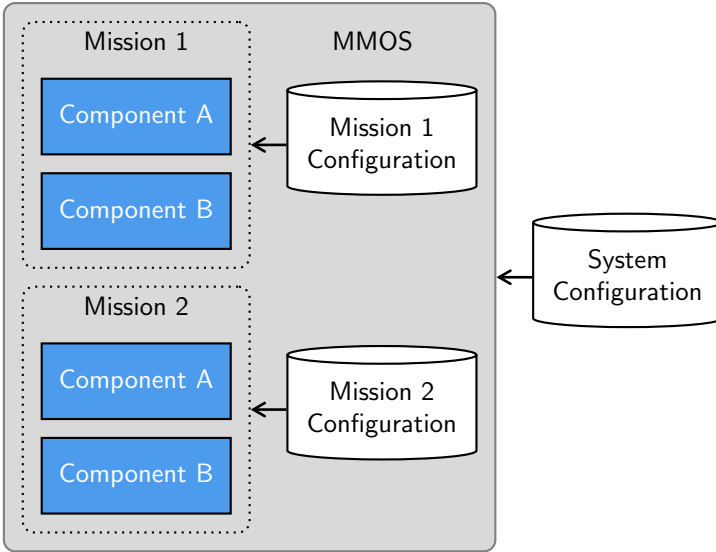


Figure 5.7: Configuration of the MMOS - Components A and B must be configured for their respective mission, as well as the MMOS itself must be configured in order to operate multiple missions. It is important to know that both components A and both components B are identical pieces of software, which only differ by their individual mission specific configuration.

sometimes even programming skills. The problem of course scales with the size and the complexity of the OS, and the number of operated missions.

Goal of this work is to close the gap between existing concepts for mission specific configuration and lacking concepts for a neat system configuration. An approach is illustrated in figure 5.7. While certain generic elements are configured for their respective mission as usual, the entire MMOS shall be set-up first by means of a standardized system wide configuration scheme. That one shall cover for example the definition of the operated missions, the instantiation of the various MMOS components, and the registration of these components within the system.

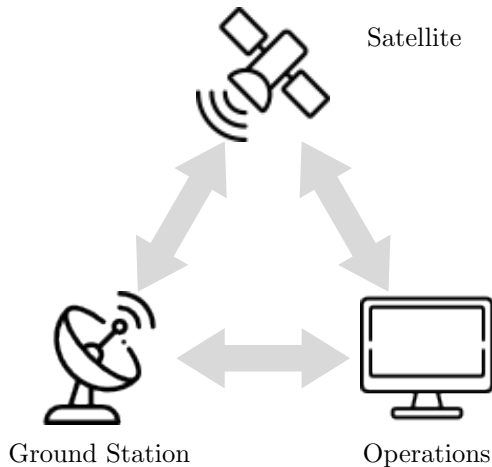


Figure 5.8: Interacting Systems of a Space System

The process of system configuration can also be considered from a different perspective. If mission operations is regarded at holistically, it is nothing more than a collaboration of three different systems: the satellite, the ground station, and the OS (fig. 5.8). What system configuration shall achieve is a description of that trio within a SSOT. That source of truth shall be the System Information Base (SIB) (sec. 5.3.3.1).

5.1.2.3 Recoverability

If one of the systems in figure 5.8 is rebooted or power-cycled, the user has a valid interest that it is returned into an operational state automatically. For instance, if the satellite is rebooted, reconfiguration is usually achieved by means of a persistent system state vector in the on-board computer, and some form of reconfiguration unit ([48], p. 323).

A similar behavior is expected by the MMOS as well. The main difference between the OS, the satellites, and the ground stations is, that the OS is in charge of the other two. This means that the

system states of satellite and ground stations must be mapped by the OS. Upon re-launch of the OS these states must be restored as well.

An example where this has not been implemented is the on-board queue feature of *SCOS-2000*. At most satellite systems, the number of time-tagged commands that can be buffered in the on-board computer is limited. A means of keeping track of that number is the *SCOS* on-board queue display. However, if *SCOS* is rebooted, that list gets lost and an operator has no information about the remaining commands in the on-board queue.

As long as the state of the operated system is not restored within the OS, (automatic) operations is impeded. So the problem of recovery is quite similar to the problem of configuration, in the sense that the MMOS itself must be restored as well as the mapped state of the systems it interacts with. Consequently, the state of the operated system shall be persistently tracked and serialized within the MMOS.

5.2 Domain Analysis

In accordance with the previously defined quality requirements, which specify *how* the system shall be designed, the outcome of a domain analysis is the specification of *what* needs to be implemented. Scope of this activity is the creation of domain models, which constitute the global system architecture. In a first step this means the definition of the various subsystems and the specification of their functional extends. In the following, the subsystem components can be defined in detail.

5.2.1 Space System Breakdown

The MMOS is part of a *Space System*. A concept for the breakdown of such a system is provided by the European Cooperation for

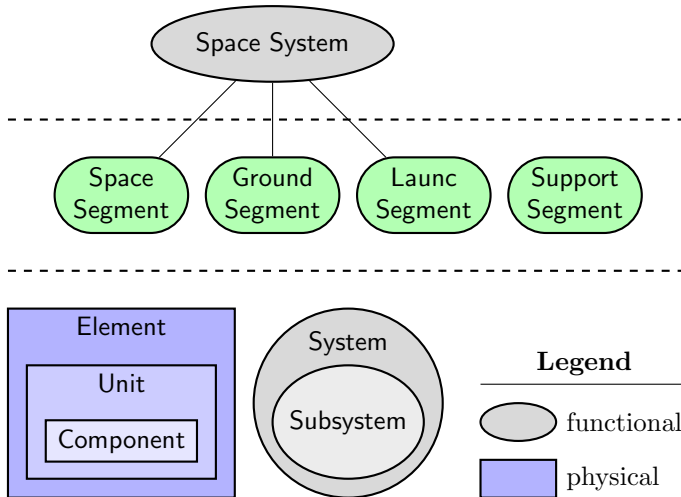


Figure 5.9: Space System Breakdown, following [37]

Space Standardization (ECSS) [37]. ECSS-S-ST-00-01C decomposes a Space System as shown in figure 5.9.

According to the standard, a Space System is broken down into three different segments, the *Space Segment*, the *Ground Segment* and the *Launch Segment*, where the Launch Segment is out of the scope of this work though and will not be discussed hereinafter.

The Space System is complemented by the *Support Segment*, featuring elements for the support of the mission, but which are not necessarily part of the mission itself. This can cover for instance: relay satellites, training and test facilities, or Mechanical Ground Support Equipment (MGSE) ([37], p. 59).

A segment describes a physical assembly of elements. For instance, the Space Segment consists of satellites, which in turn are assembled together from components. The functionality of a segment is summarized in systems and subsystems. Technical components and units are usually associated to certain subsystems, but a one-to-one allocation is not always possible.

5.2.1.1 Terminology

ECSS-S-ST-00-01C defines a large number terms to describe Space Systems. A selection shall be introduced in the following, as these terms are used repeatedly in the subsequent architecture description.

Ground Segment

Basically everything on ground that is directly involved in the conduction of the mission can be associated to the *Ground Segment*. From a functional perspective, the two prime systems within this segment are the ground station network (system) and the OS (fig. 5.8). This is of course only one way of decomposing the ground segment functionality. For other terminologies and system definitions it is referred to literature.

Element

According to the ECSS, the term *element* refers to a “combination of integrated units, [and] components [...]”. An element fulfils a major, self-contained, subset of a segment’s objectives.” ([37], p. 9) E.g., the developed MMOS software is an element of the Ground Segment.

Unit

Following ECSS, a *unit* describes a self-contained assembly of components for the fulfillment of a specific function ([37], p. 9). Within the MMOS the term is used to describe an assembly of software components for a certain subtask.

Component

A *component* is a self-contained piece of software for the fulfilment of a specific “function that can be evaluated against expected [...] requirements” ([37], p. 9). An important characteristic of a component, compared to a unit, is the fact that it cannot be disassembled without destroying its functionality. An example of an MMOS

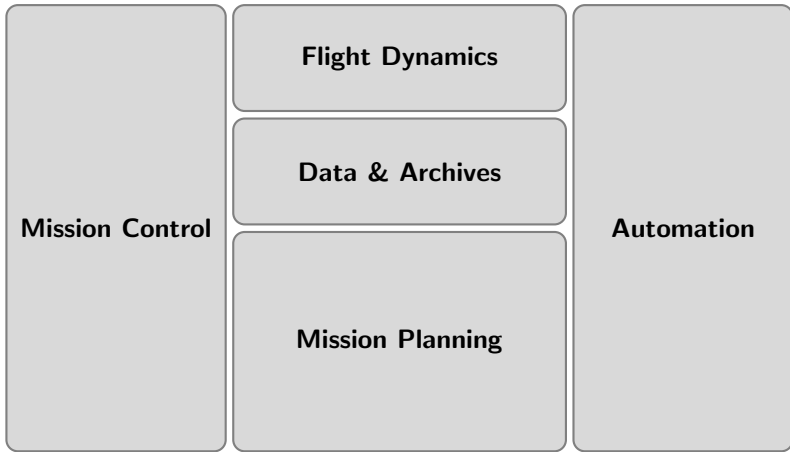


Figure 5.10: MMOS Subsystem Overview

component is the Mission Planning Tool, which will be introduced comprehensively in the later course of this work.

System

The ECSS defines a *system* as a “set of interrelated or interacting functions constituted to achieve a specified objective” ([37], p. 9). So the MMOS (as a system and not an element) is constituted by the sum of functionality provided by its components.

Subsystem

A *subsystem* is considered as a subset of the system functionality. According to ECCS a subsystem can be distributed over several segments. This will be neglected though.

5.2.2 MMOS Subsystems

Figure 5.10 displays the subsystems of the MMOS. The naming convention follows state-of-the-art operations concepts.

To this point of the elaboration that subsystem overview is still gray and empty. This will change in the later course of this work, when the subsystems in figure 5.10 will be introduced incrementally. In doing so, it will be discussed how the respective systems are commonly understood, what functionality they usually provide, and what is done fundamentally different in the MMOS.

Furthermore, in section 3.1 a series of paradigm shifts in mission operations were introduced. The following discussion will also show how and where these are reflected in the design.

5.2.2.1 Mission Control

The MCS is probably the first system one thinks about when it comes to space mission operations. One of the most famous control facilities is the NASA Mission Operations Control Room (MOCR) [46], known from the Apollo missions.

Since the early days of space flight, the MCS has been the basic system behind such a facility. Before automation was applied to mission operations, flight procedures were prepared manually in advance and uplinked by TC via the MCS. In turn, TM was received and the calibrated data needed to be evaluated by controllers. Based on that evaluation the controllers used to make recommendations to the Flight Direction for the further planning of the mission.

That procedure still applies to modern space mission operations, especially during Launch and Early Orbit Phase (LEOP) and other critical mission phases. During these phases the spacecraft is examined quite extensively in order to detect malfunction, and thus to prevent the mission from being jeopardised, or even lost.

The moment these critical mission phases have been passed, and/or automation is engaged to the operations process, the focus is shifted from single system metrics towards higher-level mission goals. That is when the MOCRs are getting empty again. During routine operations, a control room can be staffed with just a single controller.

Current MCS software, such as *SCOS* or *CCS5* [50] are primarily designed for these critical mission phases. In the later mission phases a major portion of their functionality is either covered by other units or remains unused. The functional scope of modern MCS usually covers the following items:

- **Telemetry** reception, verification, and the display of data, either in alphanumerical or graphical form.
- Release and verification of **telecommands**. The transmission is usually verified at multiple stages, such as: release, ground network transmission, reception, and execution.
- **Data archiving & distribution**: MCS usually feature an internal database for a later evaluation of the received data.
- **Event** services enable the user to trace incidents in the operated system.
- A **user management** allows to associate rules to different users of the system. E.g., command release is only allowed to someone with the appropriate rights.
- Some MCS provide means of **on-board software maintenance**, such as binary or patch upload.
- **File exchange**, if supported by the Space Segment.
- **External interfaces** allow for the connection of complementary tools. An example is the Manufacturing and Operations Information System (MOIS) [100] for the generation and execution of procedures and command sequences. ([96], p. 3 f.)

The extensive functionality and the complex architectures of current MCS software generally complicate their multi-mission use. Designed for the use in big agency MOCRs, they are usually implemented as client/server system with an own configuration, an own

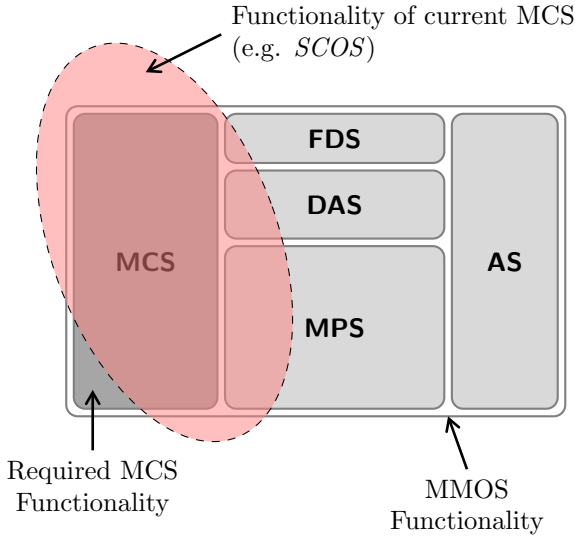


Figure 5.11: Comparison of Functionalities - Current MCS provide functionality, which exceeds the requirements of the MMOS MCS. Qualitative display only. The area sizes and overlaps have no meaning in terms of the portion of functionality to be implemented by the individual subsystems.

user management, and comprehensive internal data handling mechanisms [96]. All of this makes such systems complicated to instantiate and to maintain.

In conclusion, these systems are hardly to integrate into an architecture as of the MMOS. This will become even more evident in the later course of this work.

A solution to this problem is to implement the MCS software by a set of microservices, and to reduce its functionality in a way that it complements the overall functionality of the MMOS (fig. 5.11).

Mission Control Tool

The Mission Control Tool (MCT) shall be the central component of the MCS. Table 5.4 shows that its functionality is radically reduced, compared to current MCS solutions. This shall not mean that the

Table 5.4: Functionality of the MCT compared to other state-of-the-art MCS software [96, 125]

	MCT	<i>SCOS-2000</i> (r 4.0)	<i>CCS5</i>
Network and Connection Status	✓	✓	✓
TM Reception	✓	✓	✓
TC Transmission & Verification	✓	✓	✓
Internal TM Data Management	-	✓	✓
(Synoptic) TM Display	-	✓	✓
External Interfaces	✓	✓	✓
Scripting Interface	-	-	✓
Dedicated User Interface	-	✓	✓
User Management	-	✓	✓

MMOS is lacking of these functionalities, they are just implemented by other components, rather than the MCT.

Basically everything is *not* implemented by the MCT, which is considered not to be an MCS specific functionality: e.g., the user management. Without any doubt, a system like the MMOS needs a user management to grant certain rights to single persons or groups of people. The right of accessing the MCS and/or to send commands to the satellite is one of these permissions. These however will be granted by a central user management, and not by the MCT itself.

The MCT, like any other MMOS component, is implemented as a microservice. Users, granted with the appropriate right, are allowed to access the MCT via a Graphical User Interface (GUI). Like the central user management, that GUI is a system wide application and no specific feature of the MCT.

A spacecraft controller of course is interested in satellite TM. Various means of displaying that TM are implemented in the GUI. However, the group of people interested in satellite data does not only consist of satellite controllers, but other stakeholders too (e.g. scientists, payload specialists, engineers or mission planners). In order to satisfy the individual requirements, and to avoid a double

implementation, the functionality of accessing and processing the satellite TM is implemented by a dedicated subsystem, which will be described in section 5.2.2.2.

With a massively reduced functionality, a generic and lightweight MCT can be realized. As shown in table 5.4, the MCT features are limited to the pure task of TM reception and TC generation, transmission and verification.

Upon TM reception, the MCT shall be capable of extracting the satellite data from the received packet and forward that data to the Data and Archives System (DAS) for further processing. In turn, the MCT must be capable of composing and sending TCs to the satellite. Consequently, the tool must be capable of being configured to the protocol used by the satellite, e.g. the Packet Utilization Standard (PUS) (sec. 5.3.3.2).

The tool shall allow for two different kinds of operation: the manual uplink of commands and the automatic commanding. Both operating modes require that the MCT provides a so called *command interface* (sec. 5.3.2.1) by means of which abstract commands can be fed into the tool for TC composition (fig. 5.12). That interface shall further enable the commanding entity (e.g., a human controller or a software component) to follow the transmission process and to intervene if necessary. This also requires that the MCT provides uplink status information and information about the ground network connection status. Based on these meta information, a human controller for instance can decide whether to continue or to abort the uplink, or to try a re-send.

As indicated above, the MCT shall be a generic software component. Each individual satellite, operated by the MMOS, is commanded by an individual MCT. This means that in case a constellation of n satellites is operated, n MCTs need to be instantiated and each one must be configured for a specific satellite.

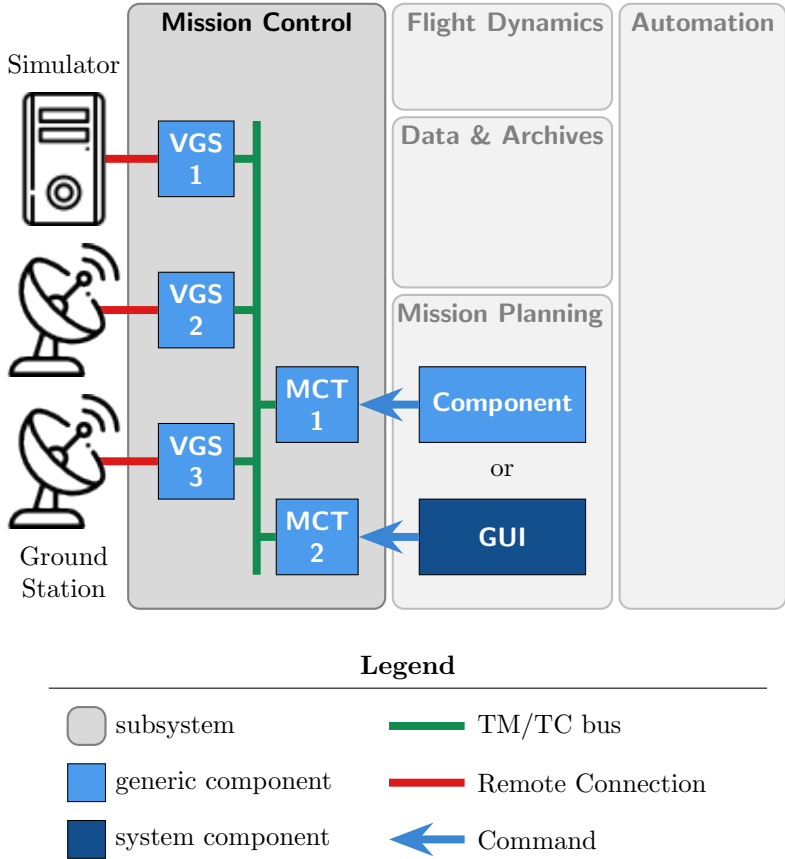


Figure 5.12: MCS setup from a data perspective - Individual MCTs for each operated system compose the TCs and receive TM from that system. The data is routed between the MCS and the various ground stations via the Virtual Ground Stations (VGS). Each ground station is addressed by an individual VGS.

Command vs. Telecommand

Here, the difference between a *command* and a *telecommand* (TC) shall be pointed out once again. All commands ever to be sent to the spacecraft originate from a schedule managed under the Mission Planning System (MPS) (sec. 5.2.2.5). Commands are added to the schedule either manually via a GUI, or automatically by planning components, which will be introduced in the later course of this work (fig. 5.12).

Commands in that schedule are protocol agnostic. They contain just as set of parameters. Purpose of the MCT is the conversion of those command information into real TCs. All the required protocol specific meta information is managed by the MCT and obtained from a central database called the MIB (sec. 5.3.3.2).

After composition, TCs are released via a microservice called the Virtual Ground Station (VGS).

Virtual Ground Station

Before a TC is uplinked, the respective data packet must get routed to the used antenna via the ground station network. Within the area of satellite operations a variety of standards and network protocols have been established:

- ESA Space Linke Extension (SLE) - Within this framework two network protocols are commonly used:
 - ESA Network Controller and Telemetry Router System (NCTRS)
 - ESA Network Interface System (NIS)
- Zodiac Cortex Protocol

At this level, information is usually transferred in frames. For testing and pre-launch operations, also some packet based protocols have been established, such as EDEN and CNC [125].

It is expected that an MMOS supports a variety network protocols. This either means to make the MCS protocol agnostic, or to allow switching the network protocol during runtime. Since the various network protocols usually work fundamentally different, both solutions were considered impractical. Designing the MCT to cope with all of the available protocols, would have made the software unnecessarily complex.

The solution to this problem is what is called the VGS. These generic software components are virtual representations for the various *real* ground stations, that are registered at the MMOS. Each VGS is configured for the protocol the respective ground station uses, and thus intercepts all connectivity issues for the MCT. Between VGS and MCTs TM/TC data, acknowledgment data, and link status information are routed via an internal bus (fig. 5.12).

By means of this approach, the system can dynamically switch between the used ground stations without the need of an agnostic software design, or the need of a reconfiguration during runtime.

Every endpoint in a connected network that implements one of the supported protocols can be addressed by means of a VGS. So, instead of a ground station, the MMOS can also be connected to a flatsat, a system test bench, or a satellite simulator (fig. 5.12).

5.2.2.2 Data & Archives

The internal management of the satellite telemetry is carried out by the Data and Archives System (DAS). Every mission OS features a subsystem like this. At first, data archived in such a system is a complete history of the satellite TM. That TM contains raw, unprocessed information from the satellite system such as house-keeping data, events etc. The gathered information can be queried and returned upon request for further evaluation. Beyond that, the archived data is also the basis on which higher level data products can be generated, for example scientific data, images etc.

The component which actually accesses and manages the databases is the TM Back-End (fig. 5.13).

Derived from the global demand for consistency, the most important DAS requirement is that every data point in the archive must be uniquely identifiable, e.g. by the Spacecraft ID (SCID), the parameter ID and a timestamp. To avoid any data corruption, the TM Back-End must have the sovereign authority on the archive, and neither another component nor entity must be able to write the underlying databases.

In addition to these fundamental requirements, the DAS must cope with some special boundary conditions and an increased performance demand. That is because of the following:

1. The DAS is part of a multi-mission system.
2. The DAS provides functionality that used to be implemented within the MCS.
3. The DAS is part of an automation process (fig. 5.1)

Being part of an MMOS implies that the DAS must be capable of handling data from different satellites and that the number of operated systems can change and grow. Further will grow the number of system accesses to retrieve data from the archive. These circumstances must not cause the system to fail and the managed data to get messed up. As a consequence, the data of each mission and each satellite must be stored within different, isolated sections.

The fact that the DAS shall implement functionality formerly carried out by an MCS increases the demand for subsystem performance. During live operations, received telemetry, or portions of it, needs to be displayed instantaneously. This is why the DAS must support the real-time routing of TM into the GUI. The GUI itself is a system-wide component and no inherent part of the DAS.

The third circumstance to cope with is the fact that the DAS is part of an automation process, which has been introduced in section

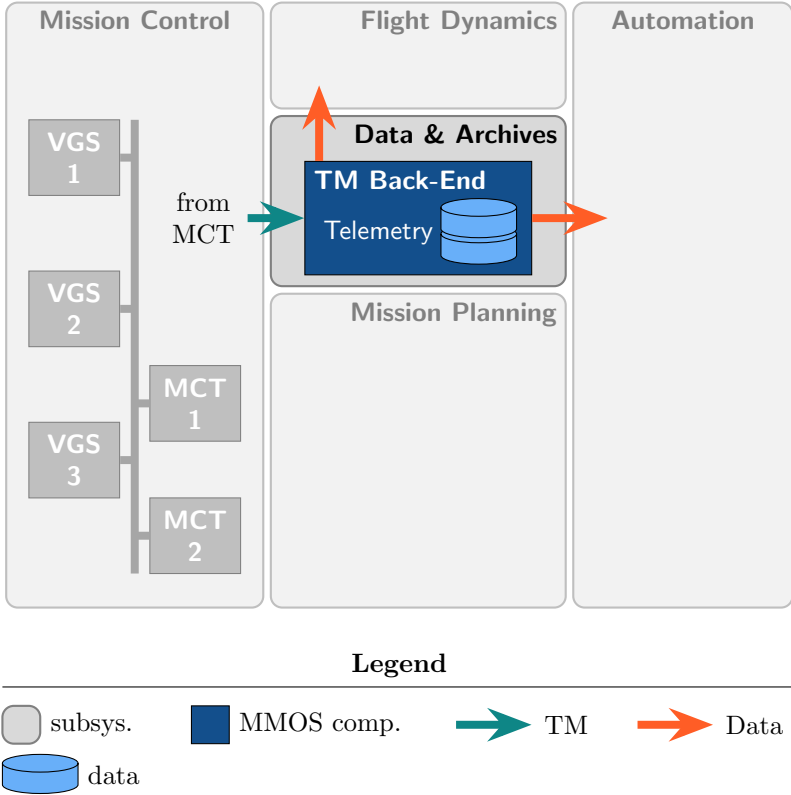


Figure 5.13: MMOS Data & Archives Subsystem - The DAS features a separate TM databases for each operated satellite. The databases can only be accessed by the TM Back-End

5.1.1.1. This means that different software components must be capable of accessing the satellite databases. Consequently, the TM Back-End implements an API called the *Data Interface* (fig. 5.13), by means of which other instances in the MMOS are able to retrieve satellite data points from the archives.

Data in the DAS archives consists of raw unprocessed telemetry, which shall be managed by means of a relational database system like *MySQL*. A benefit of such a relational database system is that it supports data consistency [95]. Even though relational database systems like *MySQL* are quite efficient and designed for large amounts of data, querying information always comes along with latencies between the request and the return of data. These latencies depend on the database architecture, the amount of data and the underlying hardware. Of course, these latencies must not impair the automatic operations process.

The implementation of databases for higher level products, like payload image data, can better be realized by means of document orient systems such as *MongoDB*, or by means of hybrid solutions. Appropriate design decisions depend on the structure of the queried data [95]. A discussion of such payload databases is out of the scope of this work though and won't be pursued in the following.

5.2.2.3 Flight Dynamics

The Flight Dynamics System (FDS) covers all those functionalities, which are related to the computation of the spacecraft motion in space. Respective software solutions are no exclusive part of satellite operations, because a lot of these features are used way before satellite launch, during mission analysis and the satellite design process.

In the scope of the satellite design process, knowledge about the orbit is important as it directly affects the thermal boundary conditions and the energy household of the satellite. If the spacecraft features a propulsion system as in most constellations or in inter-

planetary missions, flight dynamics is in charge of the maneuver planning, as well as the determination of the required Δv and the energy demand. The results of these computations have an direct impact on the mass and the structure, which in turn define the dynamical characteristics of the satellite. So the final satellite design is usually not the result of a closed solution, but a process that needs a couple of iterations.

Beyond that, the responsibilities of flight dynamics are manifold. They cover for instance:

- the model-based orbit prediction (considering disturbance, atmosphere etc.),
- the coverage analysis,
- the pass time prediction,
- the preparation and calibration of maneuvers,
- the propellant mass calculation,
- the center of gravity calibration,
- or the lifetime estimation.

Some of these activities happen outside the operations process ([128], p. 287–290). Flight dynamics computations are usually supported by commercial software solutions, which provide an extensive functionality for the various tasks described above.

Compared to all of this, the requirements of the MMOS FDS are rather limited. In simple words: The functionality of the FDS can be reduced to the pure task of determining the satellite position and velocity over time. Most commercially available solutions are way too comprehensive and thus too expensive for this purpose. Furthermore, they are hardly to integrate into the automation process as they do not feature the required interfaces.

A good alternative are open software libraries for orbit propagation, such as *SPICE*, *PyEphem*, or *Skyfield* [120, 105, 106]. The latter implement the Simplified General Perturbation Model (SGP4) [130]. An advantage of these libraries is that they enable the implementation of a Flight Dynamics Tool (FDT), tailored to the given requirements.

Within the MMOS automation process the FDT is one implementation of the data analysis stage (fig. 5.1). An important task is the determination of the current satellite orbit, based on the received satellite position data, e.g. by means of a least square method. Internally, the determined orbit can be described by means of the six Keplerian elements ([66], p. 143) or a Two Line Element (TLE), which is a standardized form of orbit description. As soon as the orbit has been determined, satellite position and velocity can be forecasted by means of one of the propagators mentioned above.

However, predicting the satellite position is usually of minor interest for mission planning. What is more interesting is the answer to the question: *When* will be the satellite at a certain location, or within line-of-sight to a certain point in space? The time period during which the satellite has a line-of-sight connection to a certain location, e.g. to a ground station, is called a *phase* in the MMOS. By means of a defined interface, called the *Phases Interface* (fig. 5.14) other components can make a phase requests at the FDT. For instance, an entity in charge of planning ground station passes for a certain satellite, can ask the FDT when within a specified time frame that satellite will pass a particular ground station. Upon such a request, the FDT returns the acquisition of signal (AOS) and loss of signal (LOS) times, which mark the begin and the end of the determined ground station passes.

Satellite missions with active orbit control, and constellations in particular, require that scheduled maneuvers are considered by the orbit propagation. After the maneuver has been executed, it is usually necessary to verify that the maneuver has been performed as

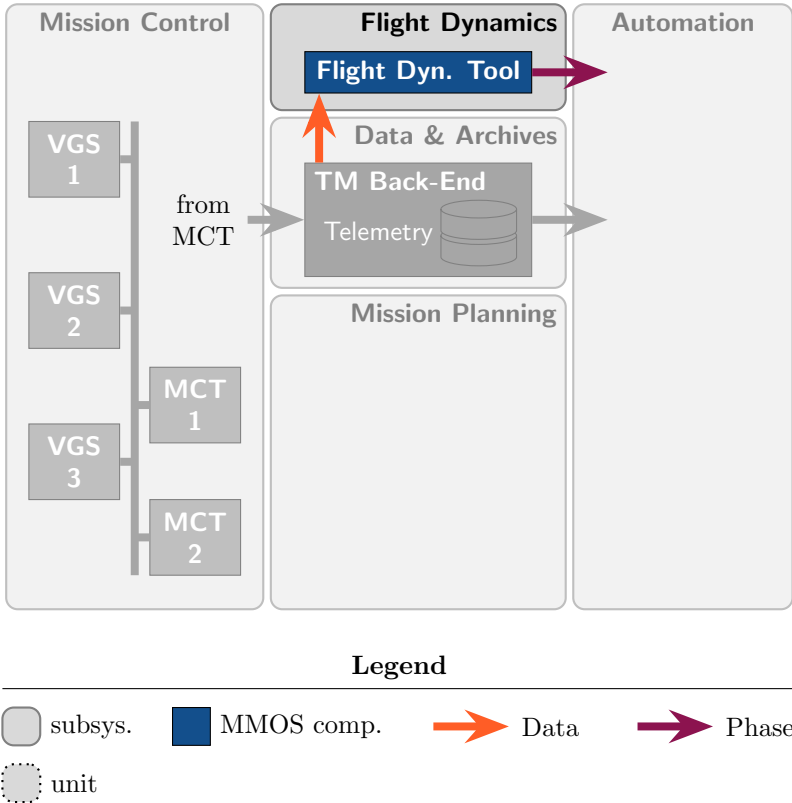


Figure 5.14: MMOS Flight Dynamics Subsystem - The central component of the FDS is the Flight Dynamics Tool. Its purpose is the orbit determination based on satellite position data, as well as the future propagation of that orbit.

planned. Hence, the FDT has to provide means of feeding satellite maneuver data as boundary condition into the propagation process. However, due to the constraints of this work, this particular feature is not discussed hereinafter.

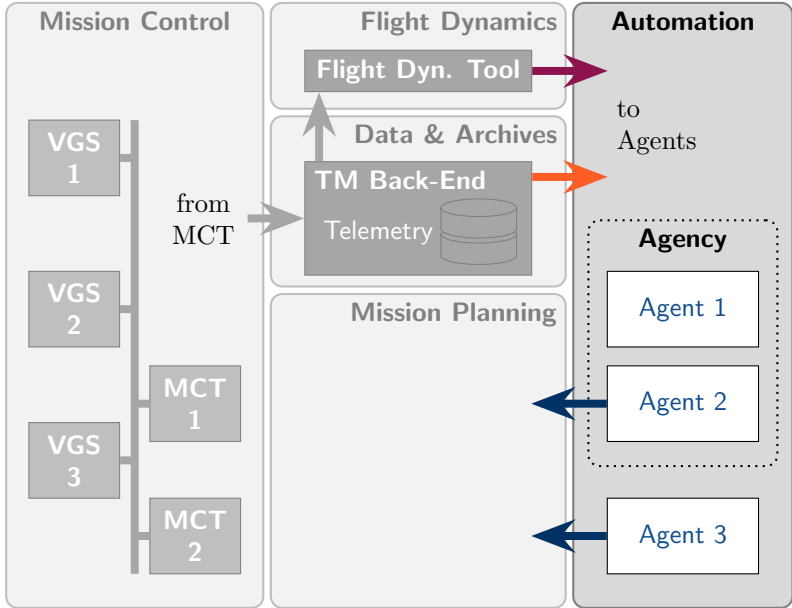
All the FDS functions described so far, shall be executed by the component instantaneously. This means that the FDT does not feature persistent databases containing the orbital data. Persistent information like current satellite TLEs shall only be issued and saved for external use, for plausibility checks, or as back-up in case the current satellite position data is invalid. This approach ensures that the orbit propagation relies on a single source of truth, which is the latest available position information from the satellite.

5.2.2.4 Automation

The Automation System (AS) should actually be called *Decision Making System*, because that is what is really happening within this subsystem. Within satellite operations the term evolved historically, and is insofar justified, as the implementation of the decision making is usually the last missing step on the way from a manual to a fully automatic system.

The process of decision making is implemented by so called *Agents*. What is an Agent? The term has been used before during the introduction of the applied automation taxonomy in section 5.1.1.1. It refers to all software components, which are no part of the baseline MMOS, and which need to be implemented for each mission individually. As mentioned earlier, a generic OS cannot provide universal decision making capabilities, because any kind of decision making requires detail knowledge about the operated system. Purpose of Agents is the generation of system *Activities* on the basis of the state of the operated system provided by the MMOS.

Depending on the application and/or the mission goals, Agents can be written to schedule different types of Activities, such as:



Legend

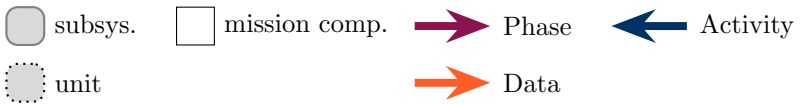


Figure 5.15: MMOS Automation Subsystem - The AS consist of several customized Agents, which are individual software solutions for the decision making process.

- Maneuver Activities
- Link Activities (Ground Station Passes)
- Scientific Data Takes
- Maintenance Activities
- etc.

The Activities in that list are merely examples though. Basically, every planned process executed by the operated system shall be related to a scheduled Activity. By providing the appropriate interfaces, the MMOS actively supports the connection of any kind of Agent initiating those Activities.

If necessary, multiple Agents can be combined to units of Agents, referred to as *Agencies*. Within these Agencies, each Agent carries out the subtask of a higher level function to be fulfilled by the Agency (fig. 5.15).

The MMOS can only support decision making processes on the basis of the data products it can provide. On the one hand, this is the raw (level-0) telemetry that can be requested from the DAS via the Data and Parameters Interface. On the other hand it is (analyzed) orbital data such as pass times provided by the FDS via the Phases interface. Of course, a decision making based on higher level data products is also thinkable. The provision of higher level data products for the further exploitation of the mission is general praxis [33]. Yet, this requires the implementation of Agents generating these data products first.

The result of the decision making process is the Activity, to be executed by the operated system. According to [49], an Activity can be described as follows.

An activity is a procedure, characterized by the defined change in the state of the system by which it is executed.

A detail definition of the Activity as used in this work will follow in section 6.2. A generated Activity is eventually handed over for execution to the operated system. The process will be introduced in detail in the later course of this work.

Functionality that Agents provide is of course highly mission specific. Consequently, Agents are individual components with a minimum of reused code. This implies that the provision of Agents cannot be the scope of a baseline mission OS. What the MMOS must provide instead, are the boundary conditions for the fulfillment of the extendability requirement (sec. 5.1.1.2), so that any kind of Agent can be attached to the system easily and conveniently for the user.

Except for the needed interfaces, the MMOS shall not make any restriction in terms of the Agent design. The interfaces classes, precisely the Phases Interface, the Data & Parameters Interface, and the Activity Interface, must be made available for the Agent developers in a widely accepted language. The same applies to any required software parts which are necessary to connect an Agent to the internal data handling mechanism of the MMOS.

Agents must be recognized and addressable by the system. Registering a component at the MMOS means that the component is known by the system, and will be executed and connected to the internal data handling mechanism the moment the system is booted. It further means assigning a *logical address* to it, by means of which the component can be identified.

5.2.2.5 Mission Planning

According to T. Uhlig et.al., the common understanding of *Mission Planning* is that it “ensures that all [system] resources are available and used to an optimal level and the goals of the mission are achieved” ([128], p. 167). This of course can mean all or nothing. In fact there is no single definition of mission planing. Specific delimitations vary with each mission (type) and the used operations

facilities. Uhlig et.al. therefore come up with an own definition, which is the interpretation widely followed by this work too.

“[Uhlig et.al.] consider mission planning as the task of preparing, organizing, and planning all relevant activities that happen during the mission, on board as well as on ground.” ([128], p. 167)

Within the domain of spacecraft operations, the process of mission planning can be implemented in various manners. Uhlig et.al distinguish between the different approaches based on their degree of automation. According to them, a mission planning system can be implemented fully automatic, as well as completely manual. A solution somewhere in between, where the operator is supported by a GUI, while internal algorithms check the consistency of the generated mission plan (usually referred to as *Schedule*), is what Uhlig et.al. consider “the most economical solution” ([128], p. 168). As examples, the authors mention the mission *GRACE* as one planned mostly by human operators, and in contrast the mission *TerrarSAR-X/TanDEM-X* featuring a fully automated mission planning [90].

The principle way of how the mission plan is generated and maintained, is one further means of distinguishing between mission planning approaches. Uhlig et.al. refer to this particular aspect as the “periodicity of the planning process” ([128], p. 169). Depending on a set of boundary conditions, the *Schedule* is prepared in different manners. The main boundary conditions are:

- the duration and the date of the scheduled activity,
- the availability of contact during activity execution,
- the periodicity of contact times,
- the criticality of the activity,
- the distance between the earth and the vehicle (transmission time)

After an evaluation of these boundary conditions, a schedule can either be prepared as a *Fixed Plan*, or be the result of a *Repeated Rescheduling*, or of an *Incremental Scheduling* ([128], p. 169).

In case of a fixed plan, a series of activities is planned beforehand and uplinked to the spacecraft eventually. This is mostly done, when operations cannot interact with the vehicle during activity execution, like in deep space missions or during descent of landing units.

Repeated Rescheduling describes the periodical revision of the spacecraft activities, and the consideration of new information by the scheduler. The reworked schedule can be the result of an optimization process, assumed that the appropriate computing resources are available.

If uncertainties make planning beyond a limited time frame impossible, incremental scheduling is usually applied. This scheduling approach starts with an empty timeline, which is continuously filled with new satellite activities. Whenever a new activity is added to the schedule, internal evaluation mechanisms must verify the consistency of the satellite schedule. ([128], p. 168 f.)

A comprehensive mission planning that meets the needs of all individual mission types equally is therefore quite unrealistic. However, a careful reader may have noticed that the scheduling approaches, described within this section, encompass aspects of the *decision making* process, and also aspects of the *action implementation* (fig. 5.1).

Within the MMOS, decision making and action implementation shall be radically isolated from each other, though. The entities in charge of the decision making are the Agents and the human mission planners. Decision making shall basically answer the question of when which activity shall be executed. As described in section 5.2.2.4, this task is highly mission specific, which is why the MMOS cannot provide a universal solution here.

The MPS shall only provide the functionalities related to action implementation. This process shall be implemented in the most generic fashion, as it is basically limited to Activity scheduling and the resolution of conflicts. The interface between the decision making (Automation) and the action implementation (MPS) is the Activity (sec. 5.3.2.2).

The moment the decision making is completely isolated from the action implementation, it does not matter anymore whether the Activity is requested by a human operator, or automatically by a software component (Agent). So the remaining question is: Which scheduling approach shall be supported. From the MPS perspective it shall not matter, whether the satellite schedule originates from a fixed plan, or an incremental scheduling. In any case, the system must allow for the addition of new Activities to the Schedule, the removal of Activities, the rescheduling, as well as the resolution of conflicts. So, the MMOS scheduling capabilities shall meet the demand of a repeated rescheduling, because this is considered the most universal one. In other words: An Activity management that supports rescheduling, also supports an incremental scheduling. And an Activity management that supports incremental scheduling can also handle a fixed plan.

Considering all of this, the MPS shall provide the following basic functionality:

- Provision of an individual (Activity) schedule of each operated system in space and on ground
- Providing means for the management of Activities requested from other entities
- Providing means of maintaining the Activity Schedule
 - Providing means of adding, removing, and the modification of existing Activities
 - Resolution of conflicts

- Resolution of the operated system state with all its resources
- Implementation of a resource model
- Implementation of a priority scheme
- Implementation of an Activity verification mechanism
 - Implementation of model for the resolution of the Activity execution status
 - Implementation of notification service for the initiator of an Activity
- Issue of an abstract Command stack, generated from the Activity Schedule
 - Generation of Command meta information (e.g. release times) based on the existing Schedule
- Providing technical means of releasing the Command stack (to the MCS)

A detail list of requirements can be found in appendix B.1.

Mission Planning Tool

The component within the MPS implementing the functionality listed above is the Mission Planning Tool (MPT).

Each system operated by the MMOS shall be represented by an individual MPT instance. Thereby, it is important that it does not matter which type of system is managed by the MPT. So, it can be a single satellite system, a ground station, a distributed system like a constellation or any other type of system compatible to the MMOS. An example setup is shown in figure 5.16.

As figure 5.16 indicates, the MPT is a generic software component. This means that always the same software is executed, disregarding the mission or the type of system represented by the tool.

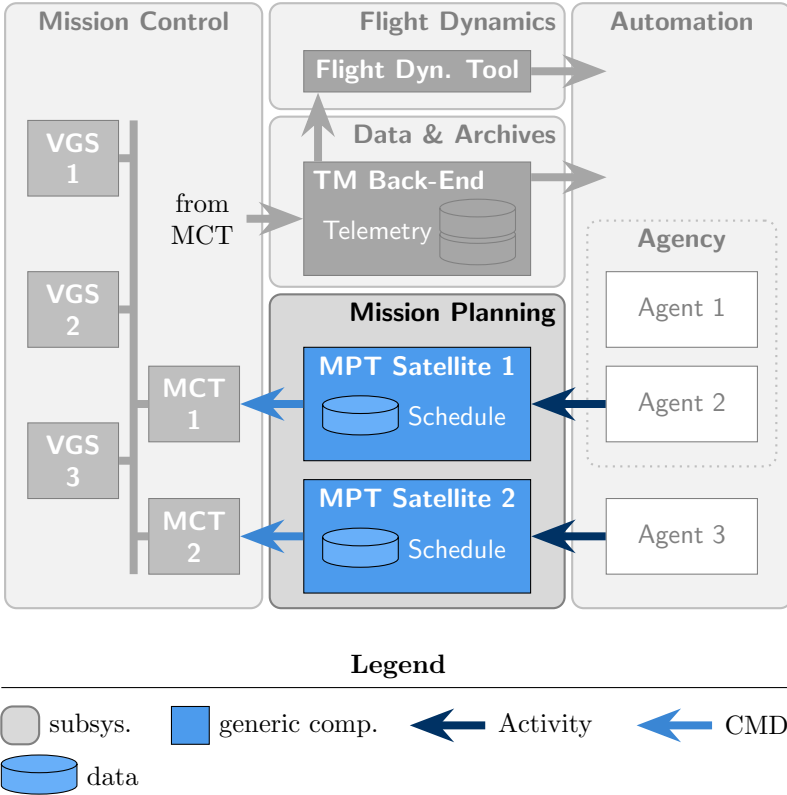
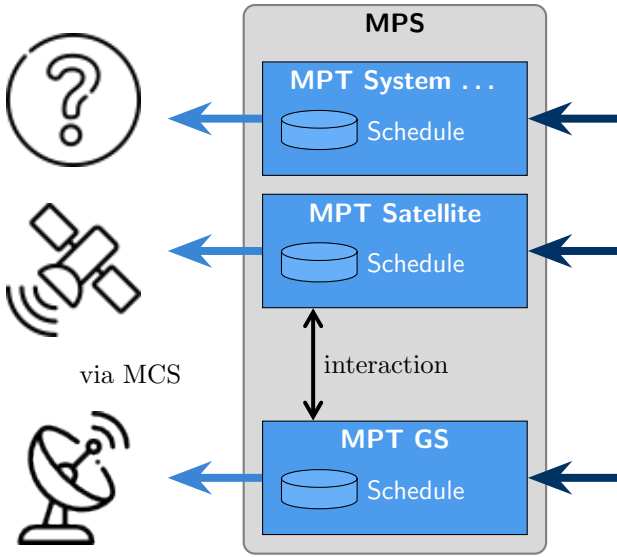


Figure 5.16: MMOS Mission Planning Subsystem - Each MPT individually handles the schedule of one operated system.

Activities are handed over to the MPT by means of the Activity Interface. Upon reception of an Activity request from an initiating entity (e.g. an Agent), the Activity is inserted into the Mission Schedule. Each MPT is in charge of a single Schedule, and each Schedule is managed by only one MPT. This sovereign authority over a Mission Schedule shall ensure that it cannot be corrupted by other entities. During runtime, the MPT performs a permanent consistency check on the Mission Schedule. Whenever necessary, activities are rejected, or just suspended and resumed later on, as will be discussed in more detail in chapter 6. The system state, which is also resolved by the MPT, is the result of this Activity schedule.

The Schedule is the basis for all activities, executed by the system. Consequently, it is also the basis for the command stack to be released via the MCT (sec. 5.2.2.1). Upon release time, abstract Commands are streamed to the MCT via the Command Interface. After TC composition and release, the transmission is acknowledged by the MCT and the status of the respective Activity in the Schedule can be marked accordingly.

As mentioned above, the MPT is the realization of the MMOS action implementation process. Following to the non-functional requirement specified in section 5.1.1.1, an operator must under any circumstances be able to monitor and to intervene the release process. The appropriate automation level according to the LOAT is D4, if the action is initiated by a human operator, and D5, if the activity is initiated by an Agent (tab. 5.3). Consequently, the MPT must implement a proper verification mechanism, informing the initiator about the current status of the Activity, and also enabling an operator to monitor and to intervene the process. The means that provides entities with the appropriate information and that grants the appropriate access rights is again the Activity Interface.



Legend

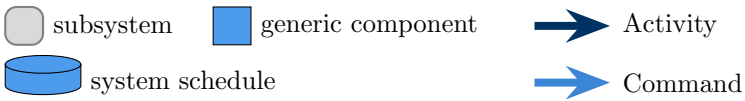


Figure 5.17: Interacting MPTs - Various MPTs are supposed to interact with each other. This is important, when resources are shared by interacting systems, for example by a satellite and a ground station during a pass.

A different MPS configuration is shown in figure 5.17. That particular image shall lead to a very important feature of the MPT.

During real operations various systems interact with each other. These interactions are not necessarily limited to one mission, but can affect systems from different missions. From the mission planning perspective, system interaction is characterized by the sharing of a resource. For example, during a ground station pass two systems communicate with each other: the satellite and the ground station. The resource both systems share is the link capacity of the antenna, which in that particular moment is not available for other missions.

If two operated systems can interact with each other, the MPTs representing them must be able to interact with each other too. Continuing the example above, this means that whenever a ground station pass is scheduled, the appropriate schedules must be synchronized and coordinated by the different MPTs.

Two conclusions can be made from that particular fact:

1. The Activity concept to be implemented must support a planning process across system boundaries. Thus, an Activity is not necessarily a single procedure, but a hierarchical process that can be decomposed into further subroutines, or subtasks. Each subtask can be executed by a different system providing its resources for the execution of the activity. The Activity concept supporting all of this will be discussed in further detail in chapter 6.
2. A sound mission planing is only possible if *all* systems related to the mission are mapped within the MPS. This means that each operated system must be represented by an individual MPT instance. E.g., if a ground station is not represented, passes using this station cannot be scheduled.

Consequently, each used ground station must be represented within the MPS, as well as all the operated satellites. A proper representation of all involved systems becomes important even more, the more

satellites are operated, and the more these satellites compete for the limited communication time [12].

Furthermore, the design of the MPT shall not make a restriction in terms of the extend of the operated functional unit. So the entity represented by the MPT can be a subsystem, a system, or a cluster of systems (e.g. a constellation). The only condition is that the functional unit can be addressed individually.

By means of this approach a holistic representation of the entire mission can be achieved as required by the underlying operational paradigm introduced in section 3.1.

5.3 Design

To this point, the required subsystem functionalities have been described comprehensively on a macroscopic level, but detail views of the system and the complete data handling process have not been presented yet. The questions that still need to be answered are: How does the proposed design intercept the complexity of a distributed system? And what actually qualifies the MMOS for a multi-mission use?

This section deals with discussion of the MMOS on system level. It shall be demonstrated, which design principles have been applied and how these promote the operation of constellations and multiple other systems in parallel. Furthermore, this section shall demonstrate how the automation process has been implemented and how all of this supports the remote control process of a spacecraft.

5.3.1 System Architecture

As discussed in section 4.3.2, the output of a design process is a software architecture. The complexity and the various aspects of software in general make it impossible to capture all characteristics of an architecture in a single display, although that is often tried

[83]. Instead, a software description must consider the recipient. In section 4.3.2 a number of common stakeholders were introduced, which normally demand for a presentation of the software from their respective view.

This is why, the architecture will be presented from two different angles. Following the previous domain description, the automation approach is introduced from a data driven perspective. A functional perspective is chosen for the description of how the system is orchestrated and how the communication between the different components is established.

5.3.1.1 Data Driven Perspective

The MMOS architecture is the realization of an automated data handling and control process. So the architecture design is ipso facto process driven. However, one of the biggest impediments on the way towards any OS is making a connection with the operated asset. This challenge still requires a data-driven perspective and a proper abstraction of the communication layers.

Figure 5.18 shows a data flow diagram of the entire system with all its subsystems and data handling components. The main direction of the data flow is indicated by the arrows, which also represent the interfaces that are defined in section 5.3.2.

What has not been mentioned before explicitly, the MMOS architecture follows two essential design principles. The first one is the principle that the system must implement a data abstraction model, so that each component or subsystem can address a particular counterpart in space.

The second principle, is that the MMOS implements a closed control loop with the satellite system. This of course is a direct consequence from the requirement for automation.

How both principles have been implemented shall be introduced in the following.

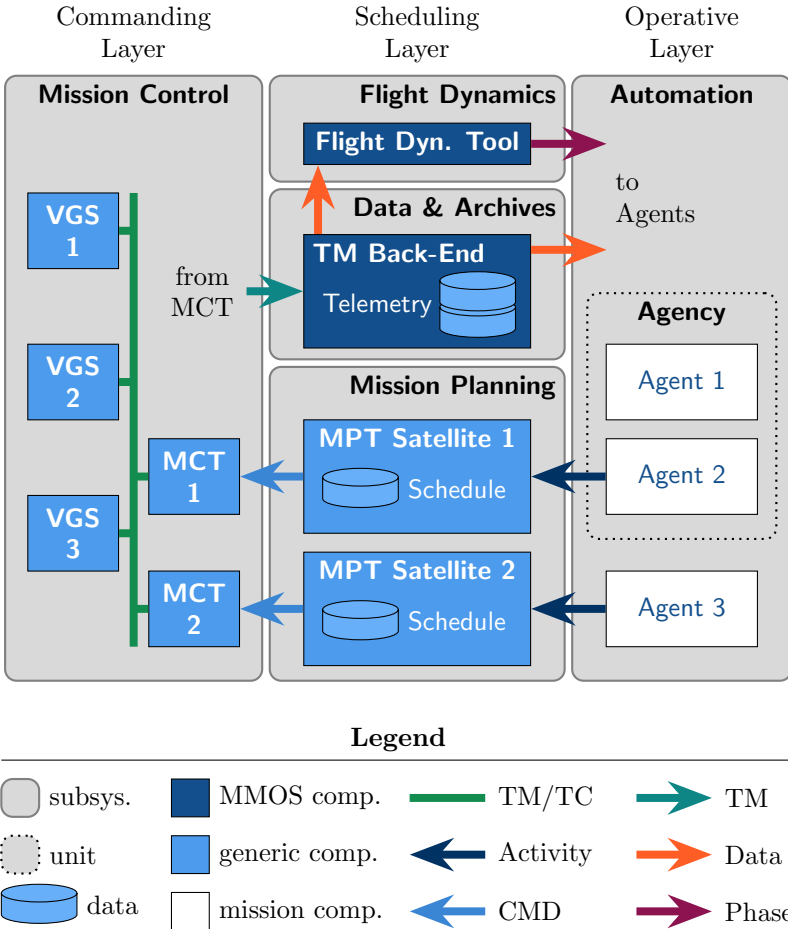


Figure 5.18: MMOS Simplified Data Driven Architecture - Entire architecture with all subsystems, components and interfaces as introduced in section 5.2.2. The figure shows an example configuration of the MMOS. The number of hosted MCTs and MPTs depends on the number of operated systems. Same applies to the VGS and the number of operated ground stations. The number of Agents depends on the complexity of the operated mission and the level of automation that shall be achieved.

The Principle of Abstraction

Complete control of a complex technical system is not implemented all at once, as the following example from aviation shall demonstrate. Before an aircraft can be equipped with a guidance system, at first a low-level controller needs to be set up, damping the natural motion of the vehicle. If that has been achieved, an autopilot can be added to maintain altitude, speed and attitude. Only if damping and autopilot work properly, high level guidance and navigation can be engaged.

The problem of communicating with a spacecraft is quite similar. Purpose of the MMOS is to establish a comprehensive and neat form of communication with the satellites so that the mission objectives can be pursued. Before something like this can be realized, both, satellite and OS, must be decomposed into layers with corresponding endpoints on each side. Here, this decomposition is referred to as *space-to-ground abstraction*. On each layer, a different protocol is applied that is used by the endpoints to exchange information. To take up the example from aviation again, only if all low-level protocols are implemented correctly, an abstract, high-level communication is possible. All of this is required for a sound automation and a subsequent development of ground-based applications for the mission.

The idea of decomposing communication between technical systems into abstract layers is not new. A widely used scheme that covers a variety of standards is the Open Systems Interconnection (OSI) model [74]. Accordingly, “the CCSDS Space Packet Protocol (CCSDS 133.0-B-1) and the ECSS-E-ST-50 series of standards address the end-to-end transport of telemetry and telecommand data between user applications on the ground and application processes on-board the spacecraft, and the intermediate transfer of these data through the different elements of the ground and space segments.” ([41], p. 8). On application layer, which is the highest layer according to the OSI model ([74], p. 30), these standards can be comple-

mented e.g. by the Packet Utilization Standard (PUS), which defines “application-level interfaces between ground and space” ([41], p. 8).

The first problem with the applications, called services, that can be addressed with PUS is that these only implement very rudimentary functionality. That is basically due to the limited performance of satellite on-board computers, and due to the consequence that the on-board software needs to be kept light and simple. So, from a perspective of an on-board software developer PUS surely addresses an application, but certainly not for a scientific user of the satellite. In a way, those PUS services can be compared with background services of a PC operating system. Users of such a PC do not want to manage these services manually. Users want the operating system to handle such services for them, while they focus on their real tasks.

The second problem is that the PUS services in the on-board software are indeed the most high-level endpoint that can be technically addressed in many satellite platforms. Any automatic process on ground that addresses the satellite on system level, has no real endpoint to communicate with. Here is where the OSI or the CCSDS abstraction models do not apply anymore.

Figure 5.19 shows into which layers communication between MMOS and satellite is broken down. The physical link to the satellite transceiver is of course established by the ground station. The communication partner of the MCT is the satellite On-board Computer (OBC), respectively the on-board software, which processes the TCs and returns TM. Up to this layer the concept is according to OSI and CCSDS models, as it describes real communication links.

The TCs, sent to the satellite are generated from the mission schedule, which is managed by the MPT. Instead of the MCT, the MPT does not address a particular component or computer, but the entire satellite system. At this point neither the OSI, nor the CCSDS model apply anymore, because the communication partner of the Mission Planning is not a real endpoint, but an abstract entity. The result of the satellite scheduling process in the MPS is

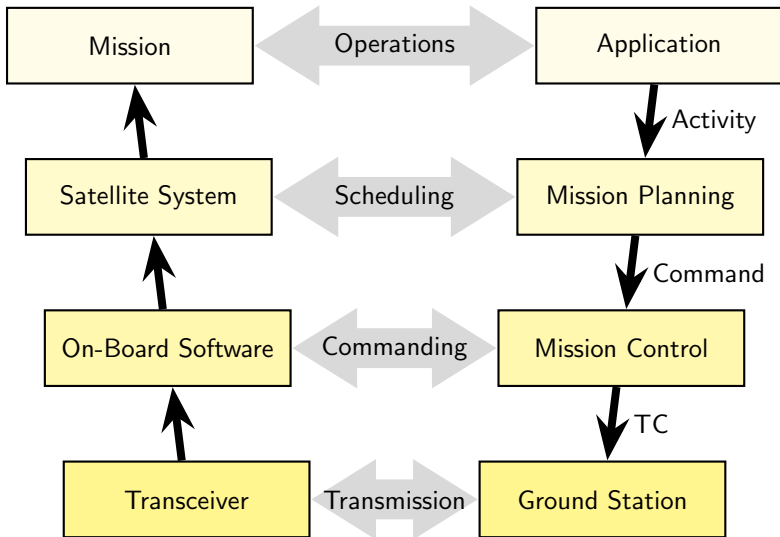


Figure 5.19: Identified Space-to-Ground Abstraction Layers for the Communication Between Satellite and MMOS - Qualitative display only. Not following the OSI model.

an alteration of the satellite system state. On ground, that system state is generated from the received satellite data.

The satellite schedule in turn is the result of a former decision making, either done by an automatic application (Agent) or a human operator. Again, this top-level abstraction describes no real end-to-end connection according to the OSI or any other model, as the application does not address a particular entity, but an abstract mission state, which must be verified by means of the processed satellite data. Maintaining the lower communication layers so that the top-level mission goals can be followed is the major purpose of satellite operations.

The Principle of Control

As mentioned in section 5.1.1.1, the MMOS implements an automatic control process, as proposed by [99]. That approach consists of four consecutive steps, the data acquisition, the data analysis, the decision making and the response action implementation. Figure 5.20 indicates how these steps are covered by the individual subsystems.

The acquisition of data is done by MCS, which is in charge of TM reception and of forwarding that data to the archives. DAS and Flight Dynamics implement the data analysis, by providing standardized means of accessing the satellite data. The outcome of the FDT is already a first level data product, generated from the satellite telemetry. Further data analysis is not implemented by the MMOS, because this is something Agents must provide. However, the prime job of an Agent is to automate the decision making process, which still can be performed by a human operator too. The result of that process is an activity to be executed by the controlled system eventually. The management of the spacecraft response action is handled by the MPT. Finally, new telemetry is generated as a result of that action.

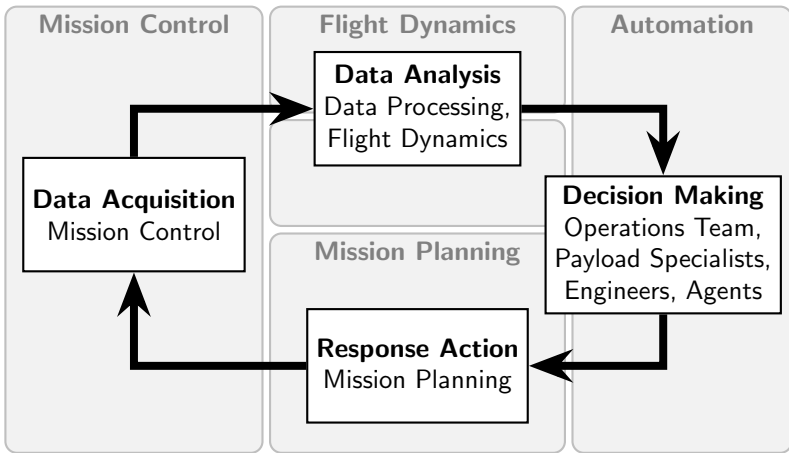


Figure 5.20: Four-stage Model of Information Processing [99], applied to Mission Operations - The figure shows which element of the MMOS fulfills which stage in the automation process.

The MMOS strictly follows the principle of a closed control loop by avoiding any kind of cross-connection and prohibiting write operations against the main control flow.

Recalling the avionics example, a mission level control cannot be realized by a single control loop, although figure 5.20 might suggest that. Instead, the control of a spacecraft is made of a cascade of concentric control loops too. The basic control, such as attitude control and thermal control, is usually implemented on board. Even more complex tasks like rendezvous maneuvering can be performed by the satellite autonomously, assumed that it is equipped with the appropriate sensors and computing performance.

Since all high-frequency control processes are implemented on-board the satellite, operations can focus on planning long-term processes. But even long-term processes can be decomposed into different high and low-level control loops, e.g. an imagery campaign. For the achievement of the campaign objectives it is normally necessary to schedule multiple data takes. Each take requires the execution of

a number of activities, for instance to change the pointing vector of attitude control and to switch on the camera.

At first, the activity execution must be verified through the evaluation of the satellite's housekeeping telemetry, in this case through the verification of the pointing vector and the power consumption of the camera. After the verification of the activity execution, the image data is evaluated. If the images are as expected, the data take was successful. The process is repeated until all the data for the imagery campaign is collected. The entire campaign can be monitored by a top-level process, or e.g. by a managing scientist.

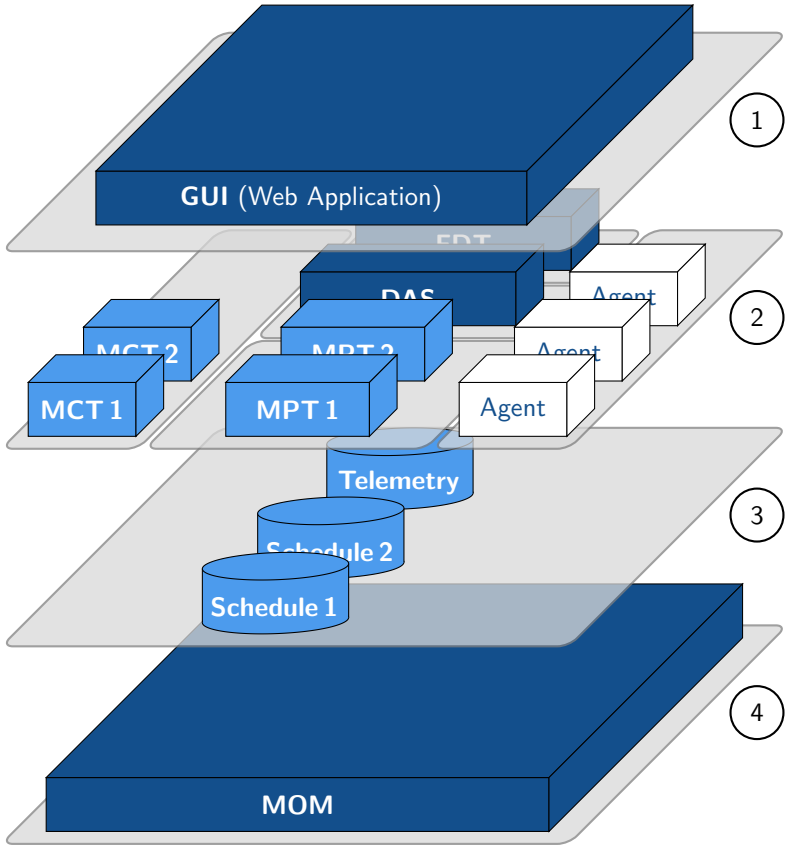
Such a process is supported by the MMOS and the Activity concept discussed in detail in section 6.

5.3.1.2 Functional Perspective

The MMOS is designed as a distributed system of individual and independent software components referred to as microservices. Each component is kept as simple as possible, with a very limited yet precisely specified scope. The rigorous distinction between the component functionalities allows that each one can be developed and maintained individually without affecting the rest of the system. A principle sketch of the MMOS functional architecture is shown in figure 5.21.

The fact that the developed OS shall support multi-mission operations implies that the system must be able to cope with growth and a dynamic adaptation according to the needs of the individual missions. This means that the entire system setup as shown in figure 5.18 can change, and that it must be possible to add new components to the system at some point. This rises the demand for scalability, especially in terms of the internal communication.

A well-known means of establishing such an infrastructure, which also comes with many other advantages, is the use of a Message Oriented Middleware (MOM). For the realization of such, reliable solutions already exist even for large systems [87].



- ① Human Interface Layer
- ② Application Layer (fig. 5.18)
- ③ Presentation Layer
- ④ Session & Transport Layer

Figure 5.21: Three-Dimensional Depiction of the MMOS Architecture - Simplified figure, VGS not displayed.

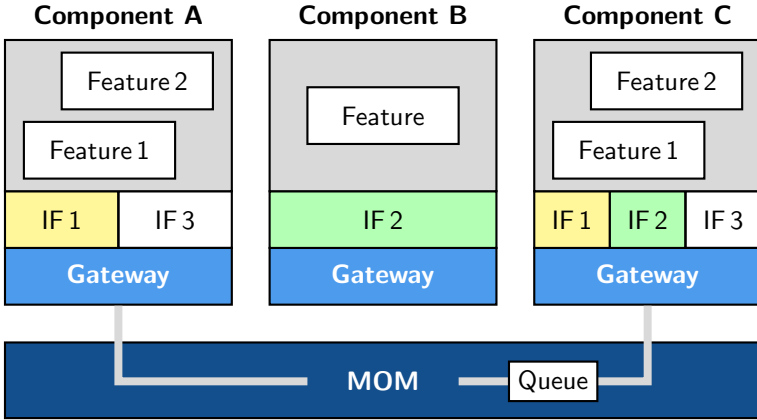


Figure 5.22: Principle Sketch of the Functional MMOS Architecture - Components communicate via message queues, which are managed by a middleware.

The idea behind a MOM is that instances in a network, which don't know each other directly, can communicate by exchanging messages in a predefined format. In the MOM, messages are routed via queues only by means of the recipient logical address (fig. 5.22). Thus, components are completely decoupled from each other and can be even moved to different locations without impairing the system functionality. The MOM takes care about the entire traffic. This covers the pure task of routing, but also the buffering of messages, and the interception of temporary connection losses. As a consequence, the system can even deal with short downtimes or load peaks [87].

Counterpart of the MOM within the component is a generic piece of software, called the gateway. It is responsible for the component authentication in front of the middleware. Whenever a new component is added to the MMOS, it is automatically registered at the middleware. By sending an authentication request to the MOM, the gateway hooks the application to the desired queues.

During operations, the gateway is responsible of composing and sending the component messages in the format that is understood by its counterpart. Consequently, the gateway is also in charge of handling messages from a queue.

Each component can implement an arbitrary number of interfaces (sec. 5.3.2). Messages received by the gateway must be addressed to these interfaces to trigger the correct functionality. This is achieved by the message format, which allows the receiving gateway to address the message content to the right interface, which in turn can respond with an answer that must be converted into an appropriate reply message.

Naturally, each component implements a different functionality. A characteristic of generic ones, such as MPT or MCT, is that they always provide the same features. A mission specific behavior is achieved through configuration. To allow that, each component is further equipped with an appropriate interface that allows loading a mission specific configuration into a component. Such an interface has not been described so far. The problem of a mission specific configuration will be dealt with in section 5.3.3.2 though.

5.3.2 Interfaces

In the scope of the prior discussion of the subsystems, the term *interface* was used quite intensively without being introduced properly.

Several interpretations of the term *interface* exist in the areas of engineering and information technology ([72], p. 574). This work follows the definition as commonly used in object-orientation. Accordingly, E. Gamma et.al characterize the interface of an object as the “*set of requests that can be sent to [the] object*” [55]. The object itself is in charge of computing information. Via the interface, information can be requested *from* the object, or provided *to* the object.

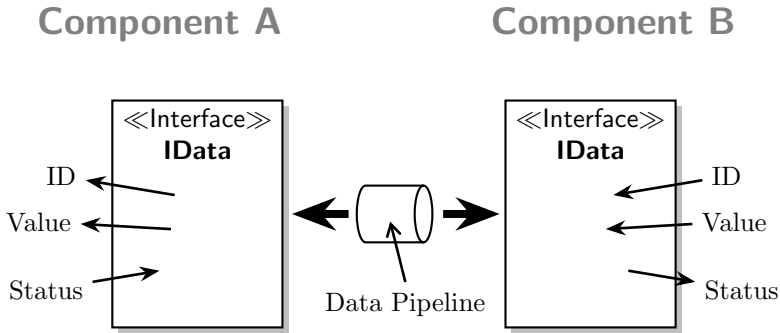


Figure 5.23: Working Principle of an Interface in a Distributed Software - Arbitrary interface object with arbitrary attributes. Convention: The preceding *I* identifies *Data* as an interface type.

In practice, there is often confusion between the concept of an interface and the concept of a *class*. That confusion is even reinforced by some coding platforms. For instance, unlike Java, the language C++ does not semantically differentiate between the two. In C++ an interface of an object is realized by the public inheritance from another class, which declares all the public virtual methods for the inheriting object [55].

Interfaces are implemented in almost every software. For distributed designs however, they have an increased value. Figure 5.23 illustrates the concept as applied within the MMOS. Two communicating components A and B exchange data. During runtime, that data is represented by interface objects within the components. Since the two do not share the same memory, or are even located on different computers, means of exchanging the data must be found.

A widely used approach is to implement the interface object as a serializable object (sec. 5.3.2.5). Once the object is defined by means of its attributes, it is decomposed into a binary stream, a packet, a message, or a file that can be transferred to a different location. Af-

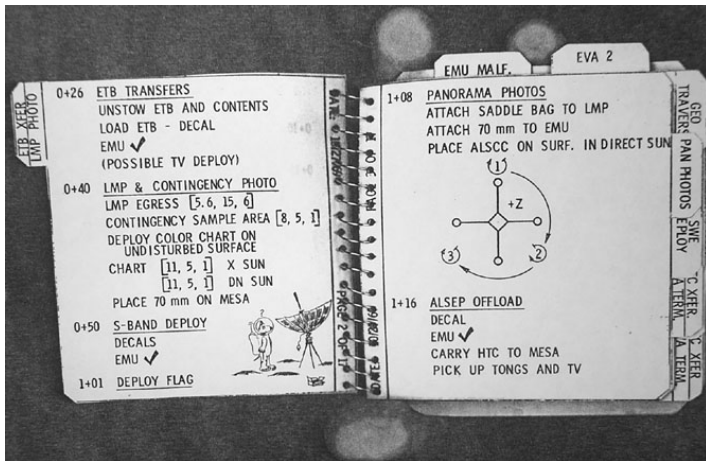


Figure 5.24: Astronaut's Procedure Journal from the Apollo 12 Mission [77] - These procedure journals contained sequences of consecutive steps to be executed by the astronauts, e.g. for extra vehicular activities (picture), or for emergency situations.

ter transmission, the same mechanism can be used to reconstitute (de-serialize) the object for further processing of the data [104].

A quality of an interface for the exchange of information between two separate applications is that it involves all data abstraction layers [74]. So, when an interface is discussed, it must be perfectly clear on which abstraction layer that discussion takes place. On lower levels, interfaces are usually understood as means of binary data transportation, or just as a physical connection. However, the higher the layer, the more interfaces are understood as mentioned initially; as a functional means of requesting and exchanging information. The MMOS interfaces introduced in the following, are discussed on such a functional level.

Command
<pre> - id : const idType - mnemonic : const char[] - description : string - executionTime : Timestamp - releaseTime : Timestamp - status : short - ParameterList : List<Parameter> </pre>

Figure 5.25: Attributes of an Abstract Command in the MMOS

5.3.2.1 Command Interface

Since the early days of human spaceflight, astronauts have used procedure journals. These journals list the steps for the fulfilment of a tasks on board a spacecraft or during an extra vehicular activity (fig. 5.24). In unmanned spaceflight the approach is quite similar. The major difference is that the steps are called commands, and that these are executed by an on-board computer instead of a human.

Object Type

Within the MMOS, an abstract command is defined by a set of attributes, as shown in figure 5.25.

- The **ID** is a unique identifier that allows to identify a particular command in the schedule.
- A so called **mnemonic** is a character string, which can be used to categorize commands.
- A **description** enables the human operator to follow the purpose of the command.
- The command will be executed by the on-board computer at the specified **execution time**.

- The command will be transmitted to the spacecraft at the specified **release time**.
- A **status** variable indicates the present transmission/execution stage of the command. Also failures are indicated.
- The command parameters are listed in the **parameter list**.

A characteristic of an abstract command is that it is designed to be human readable. Calling it *abstract* shall point out that it is protocol independent. The conversion of an abstract command into a protocol specific TC is done by the MCT the moment the command is released.

Purpose

Purpose of the *Command Interface* is to provide a mechanism for the exchange of command (CMD) and acknowledgment (ACK) information between the the mission schedule, the single source of truth for all commands, and the MCT (fig. 5.26). The spacial separation between abstract commands in the schedule and the TC composition allows for a protocol agnostic scheduling process with all protocol specific functionalities being intercepted by the MCT.

Implementation

During nominal operations the schedule is supposed to be managed by the MPS. This means that Activities (sec. 5.3.2.2) are added to the schedule, and that commands are extracted from these Activities. The MPT further controls the release process automatically. The moment release time is reached, commands are unlocked for transmission, and handed over to the MCT. The MCT instantaneously converts the commands into TCs and forwards them to the ground station.

The concept also allows users to add single commands to the schedule. Commands added via the GUI must be released manually though (fig. 5.26).

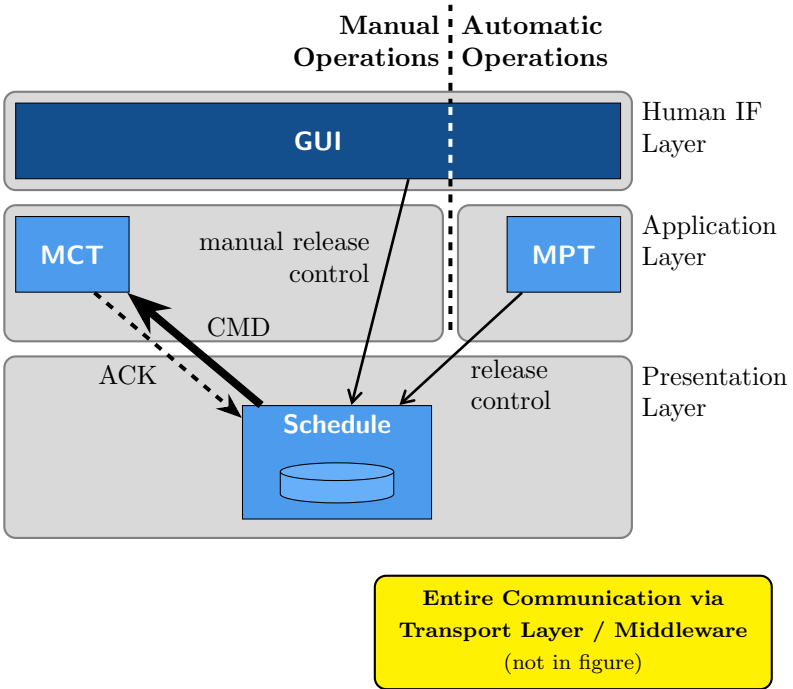


Figure 5.26: Command Interface - Depending on the operational mode, release is either controlled automatically by the MPT, or manually by the user via the GUI.

In any case, the authority over the command release lies at the MPT, or the user who scheduled the command. When a command leaves the schedule, it is processed and forwarded by the MCT as fast as possible.

The moment the MCT receives acknowledgment TM about a sent TC, that information must be associated with the right abstract command in the schedule. However, this is not a trivial process, because different protocols implement different verification schemes. E.g., the Packet Utilization Standard (PUS) has no real verification layer. Instead, transmission and execution are verified by means of a special service type [42]. Processing that kind of acknowledgment information requires a cumbersome tracking of the packet sequence count. If the MCT loses track of this count, acknowledgment information cannot be associated with the correct TC anymore.

In the setup in figure 5.26 the MCT is not only in charge of implementing such a protocol specific verification scheme, it is also responsible for the forwarding of acknowledgment information to the right abstract command in the schedule, via the Command Interface.

5.3.2.2 Activity Interface

At the end of section 5.3.2 the problem was mentioned that with every discussion about an interface the respective abstraction layer must be clarified too. However, with the *Activity Interface* discussed here, as with the Command Interface presented earlier, there is another problem.

Like the Command Interface, the Activity Interface is a means of exchanging information between different MMOS components. But that is not everything. Command and Activity are also interfaces which allow an entity on ground interacting with the operated system in space. So whenever these interfaces are discussed, it must be clarified whether the Ground-to-Ground interface or the Space-to-Ground interface is meant.

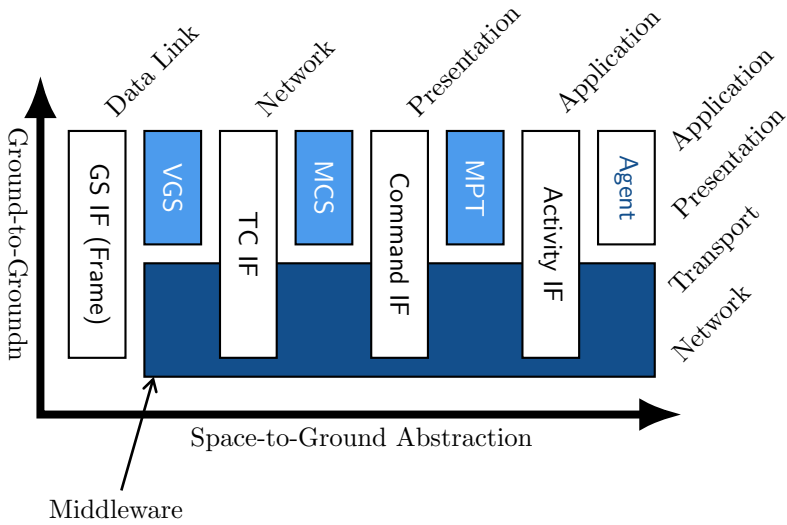


Figure 5.27: Two Dimensions of MMOS Interface Abstraction

As a consequence, the MMOS must implement two different abstraction schemes: A vertical Ground-to-Ground abstraction is needed to describe communication between different components within the MMOS. And a second, horizontal Space-to-Ground abstraction is required to analyze the communication between an MMOS component and a counterpart in space (fig. 5.27). So far, all discussions were in reference to the Ground-to-Ground abstraction.

In the Space-to-Ground scheme, the Activity represents the top-level protocol, which allows interacting with the operated system as a whole, and which shall enable the development of ground applications for that system eventually. Like in any other abstraction scheme, a top-level protocol relies on a series of lower-level protocols. In case of the Activity, this is the Command, which is converted into a TC, encoded, parsed into a bit stream, and physically transmitted via the antenna eventually. In the space segment the reverse process must be implemented of course.

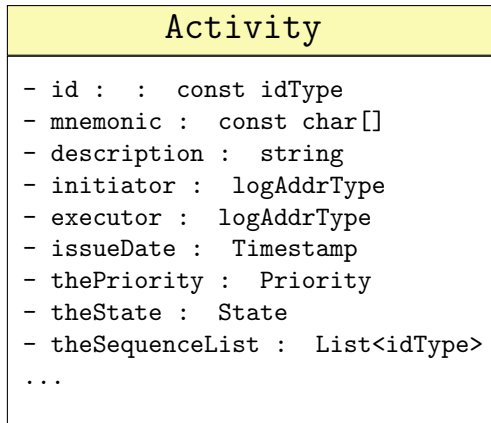


Figure 5.28: Selection of Attributes of an Activity

However, for the following introduction, that Space-To-Ground Abstraction shall be put aside, and the Activity only be introduced as an interface between MMOS components. The topic of Space-to-Ground interfacing will be revisited in section 6.1.

Object Type

As specified elsewhere, an Activity contains a sequence of commands to be executed by a system. Another quality of an Activity is that it precisely defines the change of the system state, as a result of the command execution. The concept will be introduced in detail within chapter 6. For the discussion of the Activity as Ground-to-Ground Interface, it is sufficient to regard it in a simplified fashion.

Figure 5.28 illustrates some selected attributes of an Activity object.

- **ID**, **mnemonic**, and **description** have the same function as in a command object (fig. 5.25).
- Each Activity has an **initiator**, which has defined and requested the activity.
- The **executor** is a reference to the entity (system) that executes the activity.
- The **issue date** is the date of object creation.
- Each activity has a **priority**, which is considered by the scheduling and conflict resolution process described later on.
- The **state** specifies on the hand to which stage the Activity has been executed, and on the other hand whether the Activity is currently processed or not.
- Finally, the Activity specifies the command **sequences** to be executed.

Purpose

Purpose of the Activity Interface is to provide the different entities with access to the mission schedule. *Entities* refer to all persons, software components, or systems, which are actively involved in the planning process. Entities can be human operators, Agents, or the operated system itself. Within the MPS the operated systems are represented by the various MPT instances. Each entity is also uniquely identifiable by means of a logical address.

Via the Activity Interface requests can be made at the mission schedule. Such requests can be used to add new Activities to the schedule, to alter existing ones, or to retrieve information about the Activities in the schedule.

Implementation

In the functional depiction of the MPS in figure 5.16 it looks like the mission schedule would be an inclusive part of the MPT. Also, the previous description of the MPS functionality in section 5.2.2.5 might have reinforced that impression. The implementation looks slightly different though. The schedule is actually a separate component. It manages the internal schedule database and handles the incoming requests. (fig. 5.29).

However, that schedule management, for convenience referred to just as *schedule*, is only the executing instance. All powers and decision making competencies rest with users, Agents, and MPTs. For instance, in the previous section the automatic release of commands by the schedule was mentioned. This only happens, when the respective commands are unblocked for release either by an MPT or by an operator via the GUI. The Activity interface is the means of sending such blocking/unblocking requests.

Nevertheless, each schedule is sovereignly controlled by a single MPT, which manages the activity state, resolves conflicts, determines release times, verifies the execution etc. The entire process will be discussed in detail in chapter 6. All requests necessary for

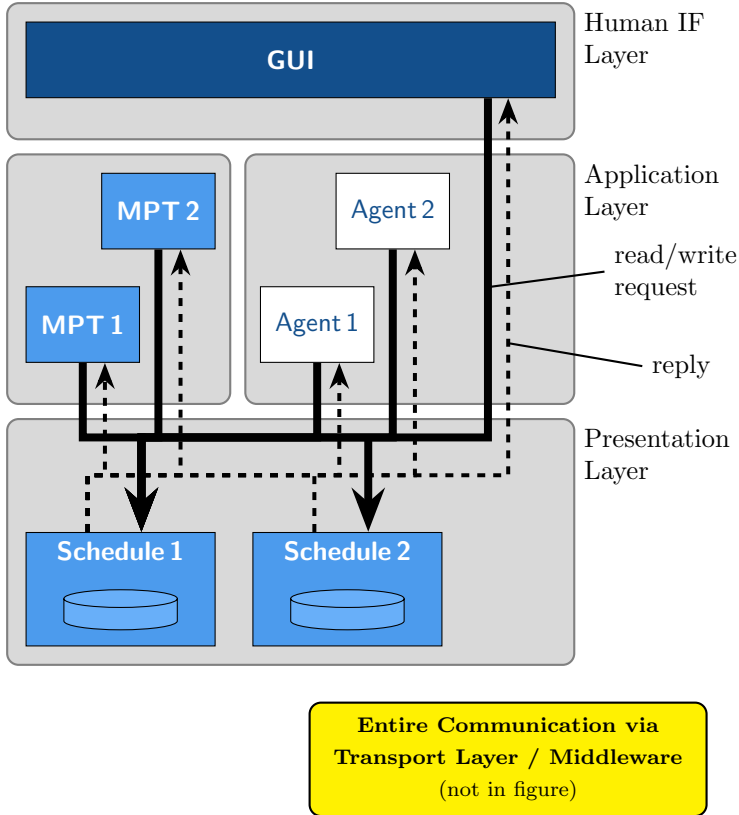


Figure 5.29: Components Implementing the Activity Interface - Via the Activity Interface entities can make requests at the schedule. Such request can contain new Activities to be added to the schedule. Requests can also be used to retrieve Activity information, or to modify Activities, assumed that the requesting entity has the rights to do so.

Parameter
<pre>- name : string - mnemonic : const char [] - rawValue : Data<ValueType> - theCalibration : Calibration - unit : string - base : ushort</pre>

Figure 5.30: Attributes of a Parameter

this task are made via the Activity Interface, while the information is persistently stored in the schedule.

According to the previous description of the MPS (sec. 5.2.2.5), only *one MPT per schedule* is allowed to modify Activities. However, the hierarchical nature of an Activity and the fact that Activities can be distributed over different systems, require that MPTs can request Activity information at any other schedule. Same applies to all the other entities. An internal access management verifies whether the requesting entity is granted with the appropriate rights.

5.3.2.3 Data Interface

In mission operations, parameter objects are used variously. They are the means of adjusting command sequences, and a means of displaying telemetry data.

Object Type

A parameter object, as shown by figure 5.30 is characterized by the following attributes:

- A **name** helps a user to follow the kind of the parameter, e.g., a battery voltage.
- By means of the **mnemonic**, each parameter is registered in the system.

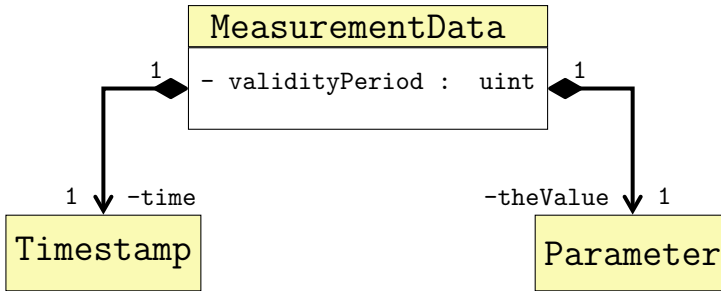


Figure 5.31: Attributes of a the Measurement Data Class

- The physical, human readable state of a variable is given by the parameter engineering value. For transmission between ground and space, this value is normally converted into the **raw value**.
- The means of converting an engineering value into a raw value is the **calibration**. The calibration can either be implemented in form of a lookup table or of a function that converts from the domain of the raw value into the range of the engineering value and vice versa.
- If the parameter represents a physical value, it must specify the according SI **unit**.
- The **base** attribute specifies the number base in which the engineering value shall be displayed. Most common bases are: binary, decimal and hexadecimal.

A particular application for a parameter is within a measurement value. A measurement data point is a combination of a parameter and a timestamp indicating the moment the value was measured on board (fig. 5.31). By means of the parameter mnemonic and the timestamp, each measurement can be identified explicitly. A further attribute of the measurement data object is the validity period indicating the duration for as long the measurement is valid.

Purpose

On ground, measurement data is gained from TM. The contained information is persistently stored in the DAS (sec. 5.2.2.2). Purpose of the *Data Interface* is to allow the retrieval of measurement data from the archive for further analysis and processing of the data.

Implementation

Measurement data is provided upon request. All components that implement the consumer side of the interface can send a request to the provider, the TM Back-End, upon which the component returns the data. As mentioned above, a specific data point is requested and identified by the parameter mnemonic, and the timestamp. The respective system must also be specified, so that the TB Back-End knows in which archive to look-up the value.

In general, data can be requested in two different manners.

1. The state of a parameter can be requested for a specific point in time. If that point lies within a period valid measurements are available, the according data is returned. Otherwise, the last available measurement is returned, with the respective attribute indicating that the returned parameter is invalid though.
2. In addition, the entire history of a parameter over a certain period of time can be queried. If measurements exist within the specified time frame, the set of data is returned.

Every component implementing the consumer side of the interface can query data like this. Examples are the GUI, the FDT, Agents, or instances mapping the TM into a system state vector, like the MPT.

5.3.2.4 Phases Interface

As mentioned in the discussion of the FDS in section 5.2.2.3, the result of the flight dynamics computations are *phases*. Phases are

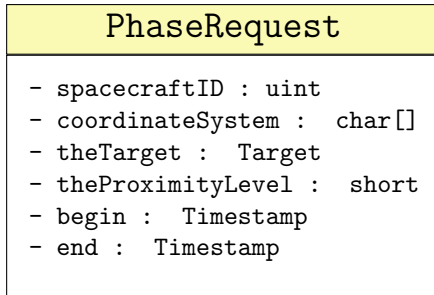


Figure 5.32: Attributes of a Phase Request

periods of time during which two objects in space are in a certain relative position to each other. A phase begins the moment the two objects have a line of sight connection and are within a specified range. Accordingly, the phase ends when the objects drift beyond the specified range again, or the line of sight condition is not satisfied anymore.

Phases are the basis of planning ground station passes, observation times, etc. Furthermore, they can specify time slots for rendezvous or maneuvers.

The means of requesting phase times at the FDT is the *Phases Interface*. Similar to the Data and Parameters Interface, phases are returned by the FDT upon request by a consumer. The attributes of such a phase request are shown in figure 5.32.

- Each request contains the **ID of the spacecraft** whose orbit propagation in the FDT shall be the basis of the phase computation.
- The **coordinate system** is the frame in which the second point (target) is specified.
- A **target** object describes the location of the second point in the specified coordinate system. The target can either be a fixed location in the frame or a moving object (e.g. described by TLE)

```
{
  'Family': {
    'Mother': 'Jane',
    'Father': 'Michael',
    'Children': [
      {
        'name': 'Philipp',
        'age': 16
        'gender': 'male'
      },
      {
        'name': 'Kim',
        'age': 12
        'gender': 'female'
      }
    ]
  }
}
```

Figure 5.33: JSON Example

- The **proximity level** parameter describes how close the two bodies must approach for the phase to begin.
- **Begin** and **end** mark the time frame during which phases shall be determined. All phases are returned that begin or end within the specified time frame.

If at least one phase falls within the specified time frame, the begin and end times are returned by the provider side of the Phases Interface. Each component that implements the consumer side of the Phases Interface, can make such requests at the FDT.

5.3.2.5 Serialization

The interfaces types introduced in this section are structured objects. These have in common that at some point they need to be decomposed, transferred, and reassembled again. The process of ob-

ject decomposition for a use somewhere else, or for memorizing an object state, is referred to as serialization [104].

A popular format for the exchange of structured objects between different applications is the JavaScript Object Notation (JSON). Although its name suggests otherwise, JSON is agnostic of the programming language [137]. It is a notation for text files containing the attribute information of the serialized objects [36, 73]. JSON has been the selected message format for the transmission of data via the MOM [5]. An example is shown in figure 5.33.

5.3.3 System Configuration

The development, the setup and the configuration of the ground software is generally driven by the space segment. That is because of the usual practice of the satellite development starting way ahead of the ground segment, which then must get tailored to the space segment eventually. Presumably, this will always be like this. The approach is insofar understandable as the spacecraft is usually the more challenging and driving part of the mission.

To this date, a vast number of satellites is still developed from scratch, supported by a variety of tools, techniques, and standards which are available these days. Nevertheless, new interfaces need to be developed or at least tailored for almost every new satellite, or at least for the payload. “The interface design for spacecraft systems and spacecraft payloads is still a manual and time-consuming effort.” ([23], p. 5)

Purpose of an interface is to guarantee a flawless interaction between ground and space on different abstract levels (fig. 5.19). By defining a standardized space-to-ground abstraction, the MMOS encounters the demand for a neat multi-level connection. However, each component implementing one of these interfaces must get configured, so that the component knows how to use the interface cor-

rectly. An MCT instance for example needs to know which TCs can be processed by the on-board software.

In satellite operations, the general method is to define the mission specific interface configuration in a persistent database. These databases are commonly referred to as SRDB or Mission Information Base (MIB). Several standards for the setup of such databases have been defined in the past, like XML Telemetric and Command Exchange (XTCE) [25], or the *SCOS-2000* standard [52], and both of them have their justification, advantages and disadvantages.

- Both standards are compliant to the CCSDS and designed for a space application.
- Using standardized databases supports the exchange of configurations between different ground systems.
- A new configuration can be loaded into the system fast.
- The configuration can be generated automatically from the on-board software.
- The application of those standards supports testing and verification.

Despite all these benefits, those MIB standards also have a major drawback. Existing MIBs are limited to command layer communication. The two mentioned standards have been designed just for the configuration of MCS, and for the exchange of configurations between different MCS. They are explicitly not designed for the mission specific configuration of higher-level components as they exist in the MMOS or any other OS. That is not surprising due to the huge variety of systems available on the market. It would not be easy to define a MIB standard that suits all existing systems equally in the mission specific configuration of their components.

The *SCOS* MIB for instance allows for the definition of TM / TC packets, parameters, events, or forms of displaying TM. For system

level operations though, a MIB must also specify metrics by means of which the state of the system can be quantified, as it must indicate system variables which have a relevance for high-level operations (e.g. GPS x, y, and z position, which can be the basis for orbit determination by the FDT).

One of the advantages of such a database concept is also its biggest disadvantage. A familiar promise that is made by almost every MIB is: *Load me into your system, and you are ready to go!* Yet, practice has shown that it is not that easy. On the one hand, the concept of persistently saving configuration parameters in a database is very convenient. On the other hand, existing MIBs merely allow for interface configuration. An OS however has other aspects too, which raise the demand for a configuration.

The setup and the configuration of the interfaces is without a doubt one of the biggest burdens for a ground systems engineer, but it is surely not the only one. An effort that is often underestimated is the configuration of the OS *itself*. In order to cope with that, a second database called the System Information Base (SIB) shall be introduced for the initial orchestration of the MMOS. The concept takes advantage from the known benefits of a standardized configuration, and thus enables the automatic launch and (re-)configuration of the MMOS.

Launch and configuration of the MMOS are done in two steps:

1. System boot, and initialization of each component by means of the SIB.
2. Mission specific configuration of each component by means of the MIB. Each mission is supposed to provide an individual MIB that can be loaded into the system.

5.3.3.1 System Information Base

The process of launching the MMOS by executing all its components and initializing them is referred to as *orchestration* in this work. Basis for the orchestration are the entries and parameters specified in the SIB .

A graphical representation of *what* is configured by means of the SIB is shown in figure 5.34. In its essence, the SIB is the implementation of the MMOS feature tree model (sec. 4.3.1.1). The database specifies which features shall be activated and it resolves dependencies between them.

In section 5.1.2.3 it has been pointed out, that a space system consists basically of three major entities: The *satellite* (mission), the *ground station*, and the *OS* (fig. 5.8). A fourth entity that has been disregarded so far are the users, respectively the operators working with these systems. Purpose of the SIB is setting up the MMOS so that the configuration reflects that *space system* (sec. 5.2.1). Therefore, it is not surprising that all these entities show up in the SIB as well.

Setting up the system itself means first orchestrating all components that are not used for one mission particularly, which are the MOM, the GUI, and all the other commonly used components. Since the MMOS itself needs to be operated and maintained too, user groups regulate by whom the system can be used and who is allowed to do what, like performing administrative activities, adding missions, and so on.

In a second step the missions are defined. Creating a mission in the database means listing all those components, which should be used by the mission. This can be an arbitrary number of archives, an arbitrary number of MCTs, an arbitrary number of Agents, and at least one MPT. As will be discussed later in chapter 6, complex satellite systems are represented by a network of multiple MPTs.

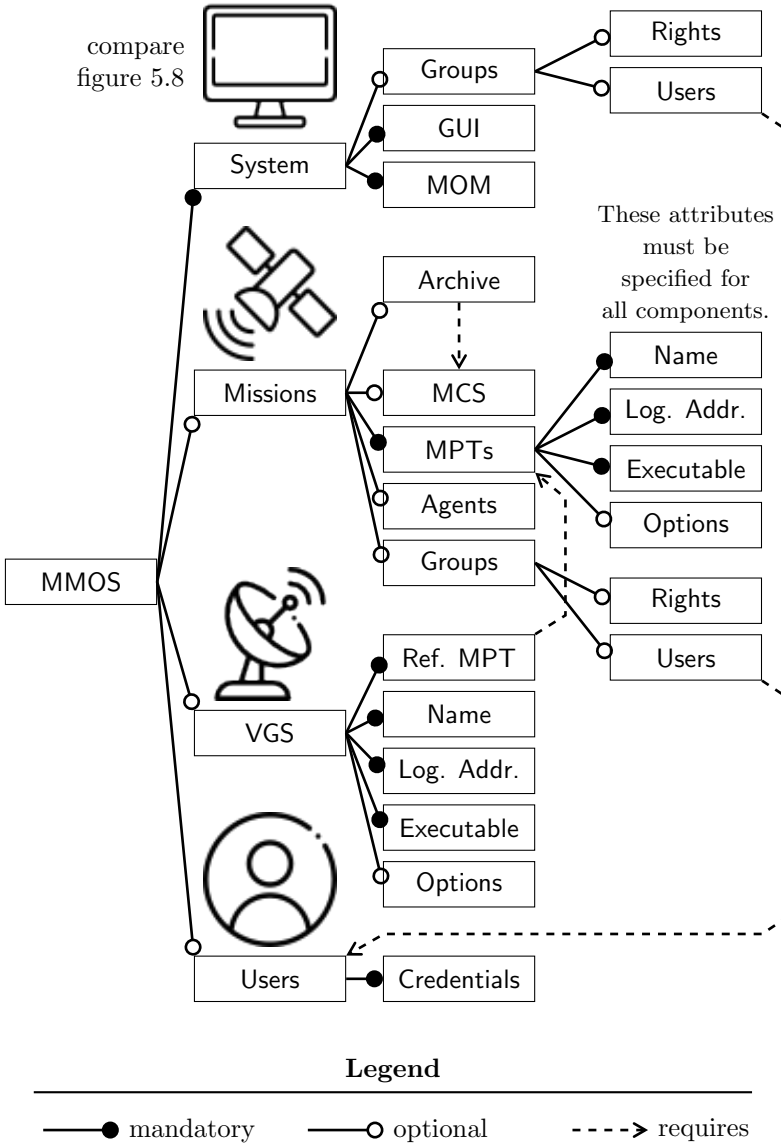


Figure 5.34: System Configuration - Simplified representation of the system feature tree as modelled by means of the SIB. The tree indicates which system features, elements and components can be executed and which attributes must be specified for the configuration of them.

Multiple MPTs then of course demand for the appropriate number of MCTs.

Each component definition must specify a name, a logical address, and a path to the executable. In case of a baseline component, the path is a default setting. It must only be adapted in case the respective component is a mission specific software, e.g. an Agent.

Every operations team consists of different members. There is the Mission Director, the Flight Director, several System/Subsystem Engineers, and others ([128], p. 42–46). These people must have different rights. Some shall be allowed to schedule activities, some shall be allowed to approve these, and some shall be allowed to send direct TCs. Therefore, the SIB supports the definition of user groups to grant people with the appropriate access rights.

The third entity type to be configured are the ground stations represented by the Virtual Ground Stations (VGS). These VGS must be defined like any other component. The major difference is that the use of a VGS is not limited to a single mission. However, ground stations are scheduled like any other system too. This is why each station must have its own MPT.

Finally, the people using the MMOS are defined in a fourth section. The users created here are the ones that can be granted with access right as mentioned above.

5.3.3.2 Mission Information Base

Purpose of a MIB is the configuration of the various communication interfaces according to the implemented space-to-ground abstraction scheme (fig. 5.19). In the past, the concept of a MIB was limited to the use by an MCS, and to a configuration up to commanding layer. MIBs were used for the configuration of TM / TC parameters, command sequences and user interfaces (displays) ([52], p. 11).

Within the MMOS, the scope of the MIB is extended in the way that it supports an additional so called scheduling layer on top of the commanding layer.

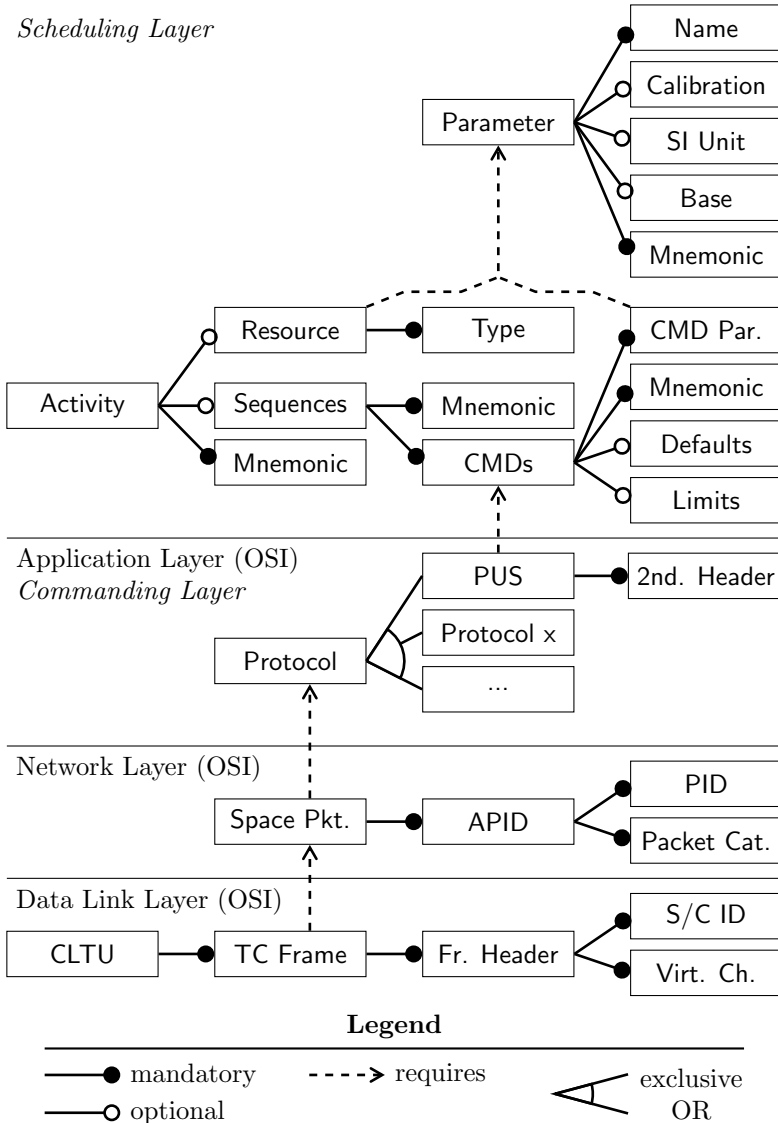


Figure 5.35: Mission Specific Interface Configuration - On each abstraction layer, certain interface attributes must be specified by the MIB. The figure illustrates these attributes. It only shows the configuration of the uplink (TC) though.

As known from other standards, uplink and downlink interfaces are configured almost independently. The breakdown of the uplink configuration is shown in figure 5.35. By using the notation of a feature tree model, the figure indicates which object types must be specified on which abstract layer, as well as the dependencies between the different elements. Up to commanding layer, the extend of what is going to be configured by the MIB is derived from the *SCOS-2000* Interface Control Document (ICD) [52].

The MMOS supports SLE, so the MIB must enable the creation of a Command Link Transmission Unit (CLTU) containing the TC frame. This also covers the specification of the coding scheme, or details about a possible encryption (not mentioned in figure 5.35).

Frames encapsulate the transmitted CCSDS Space Packets [24], which are identified by means of an Application Process ID (APID). On the next layer, the MIB defines the used application layer protocol (e.g. PUS). The application layer protocol must be specified exclusively, which means that only one protocol can be selected for one operated system.

Up to this point the extend of the MIB does not exceed the scope of a *SCOS-2000* MIB. On the next layer however, the MIB configures the system level interface. By doing this, the MIB enables a close interaction between the satellite commanding and the system level mission planning (scheduling).

The interface that allows for a system-level communication is the *Activity*. For the composition of such, the MIB must specify a number of abstract commands (sec. 5.3.2.1), and command sequences, which shall be executed during an Activity.

The successful execution of an Activity depends on the availability of the appropriate system resources. Resources are certain system parameters, which are later mapped into a state vector. For the definition of such, these parameters must be specified as well. The concept of how resources are used in the Activity scheduling process is discussed comprehensively in section 6.

Mission Planning

While the previous sections went through a comprehensive description of the Multi-Mission Operations System (MMOS), this chapter will cover detail aspects of the Mission Planning Tool (MPT) design, and describe how that component enables the scheduling of a distributed system eventually.

6.1 Value of the Mission Planning Tool

Among of all MMOS components and subsystems, why is it so important to have a closer look at the MPT particularly? To answer that question it is worth to make a step back to have an examining view at the MMOS architecture.

Figure 6.1 displays the MMOS in two different ways. The left-hand side image is a simplified version of the functional architecture as shown in figure 5.21. Reason for that kind of architecture with a central Message Oriented Middleware (MOM) was the demand for scalability (sec. 5.1.1.3), and extendability (sec. 5.1.1.2). Organizing the MMOS this way allows for the addition and the launch of different applications, and thus for a flexible orchestration of the system. From this perspective, the term *application* refers to all the MMOS components (Virtual Ground Stations (VGS), Mission Con-

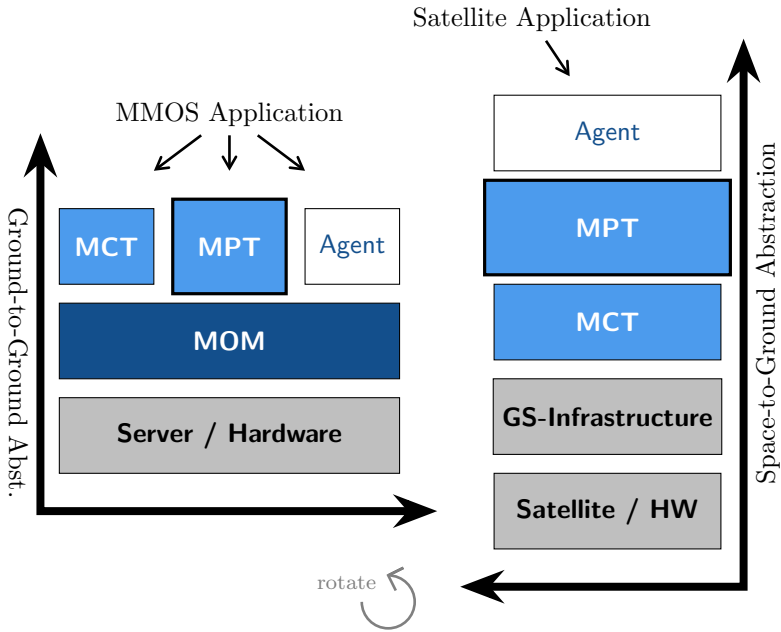


Figure 6.1: Implementation of a two-dimensional Abstraction Scheme - From an operational perspective, the MPT is the interface between mission specific applications (Agents) and the rest of the MMOS architecture. The respective abstraction scheme is referred to as *Space!-to-Grnd. Abstraction*.

trol Tools (MCT), MPTs, etc.) as introduced in section 5.2.2, which are hosted as microservices on top of the MOM.

But, neither the MCT, nor the MPT, nor any other of those microservices can be considered as an application from an operations perspective. Instead, they are just means of establishing the data connection to the satellite. An application that has a real value for an operations team is something that needs to be implemented on top of that infrastructure. This is why a second dimension to the MMOS architecture has been introduced, which is referred to as Space-to-Ground Abstraction (sec. 5.3.1.1), and which is indicated on the right-hand side of figure 6.1.

So what makes the MPT so special? The MPT is the instance that manages the satellite schedule. Whenever a *real* satellite application is written (e.g., an Agent planning scientific data takes), all Activities generated by that application need to pass the Mission Planning System (MPS). The MPT and its underlying schedule are thus the interface between all those top-level applications and the rest of the architecture. Among other things, the MPT performs schedule consistency checks through monitoring and forecasting the satellite state, it automatically controls the command release, and verifies the Activity execution.

All of this makes the MPT the heart and soul of the MMOS, which is why it shall be discussed in detail hereinafter.

6.2 The Activity

The scheduling process carried out by the MPT is based on the Activity concept. During the previous sections of this work the term *Activity* has been used extensively, without even being defined properly. A single definition of it is not trivial though, as it depends on the application and the individual perspective. In this work, the Activity is looked at from three basic angles:

- the operational perspective,
- the data-handling perspective,
- and the scheduling perspective.

Operational Perspective

Choosing the operational perspective is probably the most obvious way to define the Activity. From the operational angle, an Activity simply describes a process executed by a system or an entity, to perform a certain task. A quality of an Activity is that it affects the state of the executing system, and demands system resources like energy.

Data-Handling Perspective

From the data-handling perspective, the Activity refers to an object type, which can be used to exchange information between elements of the MMOS. Thus, it describes an interface that software components can implement to interact with the system schedule managed by the MPT. This perspective was chosen during the Activity (interface) description in section 5.3.2.2.

From a data-handling perspective, an Activity further refers to an interface between space and ground (sec. 5.3.1.1). As such, it is a means of exchanging all necessary information with the executing system. These information cover data for activity execution, as well for its verification.

Scheduling Perspective

In this chapter, the Activity is regarded at from the scheduling perspective. In this context, an Activity refers to a scheduled object, which provides all information needed by the MPT to manage, execute and verify the mission plan. The attributes of an Activity as an object within the mission schedule are shown in figure 6.2.

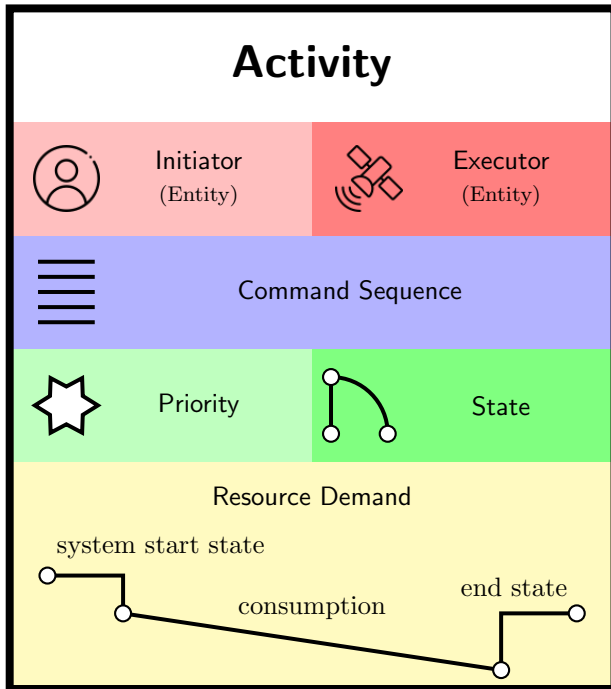


Figure 6.2: Attributes of an Activity

Each Activity has an *initiator*, a reference to the entity that has planned the Activity, and an *executor*, which is a reference to the system executing the Activity. Both attributes are important identifiers in the planning process.

The *command sequence* is a reference to the command list to be released to the executing system eventually. The states of the released commands are indicators considered during the Activity verification process.

Priority and *state* are important attributes for the resolution of conflicts, and for the scheduling process that will be introduced in section 6.2.4.

As mentioned above, the execution of an Activity is bound to the availability of system resources. The amount of the resource consumption is quantified by one or more *resource demand* objects. The conditions specified by these objects are the basis for the resource propagation and the conflict determination by the MPT, as they describe the required state of the operated system at the beginning of the execution, as well as the resulting end state. Details of the state estimation based on the resource demand are handled in section 6.2.3.

6.2.1 Inheritance

Activities are so called *scheduled objects*. As such they must provide a number of attributes, which are necessary for their identification in the scheduling process. Since activities are not the only objects which are involved into that process, an inheritance structure has been created, which several of those other objects and interface types can reuse (fig. 6.3).

Each object that is somehow handled in the course of the operations process, or is exchanged as an interface type (sec. 5.3.2), shall

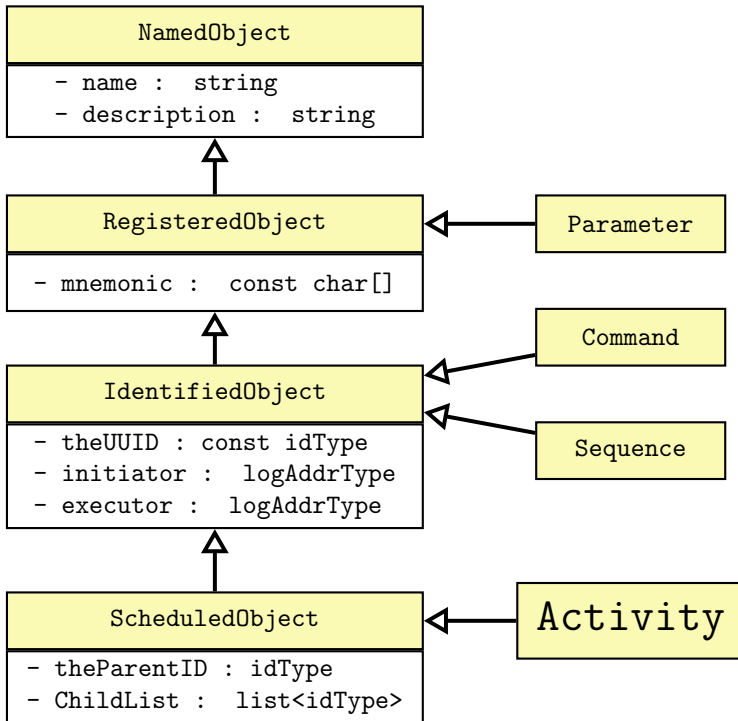


Figure 6.3: Inheritance and Serialization of an Activity Object

have a name and a description, which help a user to understand its meaning at a glance.

Each activity further must be registered at the system. Only registered objects can be instantiated and thus appear in the system schedule. A quality of a registered object is that it is defined once and then can be instantiated, or appear in various states at different times. For instance, a system parameter is defined once. Over time that parameter (e.g., a battery state of charge) will be in various states, which are recorded in several data points within the archive. An Activity is also defined once, and then can be scheduled over and over again. Registered objects are identified by a so called *mnemonic*. Mnemonics are short character arrays that also support structuring the registered objects.

Once a registered object has been instantiated it must be uniquely identified. Activities, as well as the commands in the schedule, are identified by a constant Universally Unique Identifier (UUID) [5]. That UUID is created at object instantiation and cannot be changed. Scheduled objects further specify an executor, which helps associating the object with the executing mission, while the initiator is a reference to the entity which created the object.

A special quality of the Activity as a scheduled object is the optional specification of a parent and several children, which supports nesting the Activity. The concept of nesting will be discussed in detail in section 6.2.5

6.2.2 Class Diagram

In addition to the object structure specified above, an Activity is characterized by a set of further attributes (fig. 6.4). As mentioned in section 5.3.2.2 already, each Activity object features a time stamp indicating the issuing date, and a priority that helps resolving conflicts between different Activities during the scheduling process.

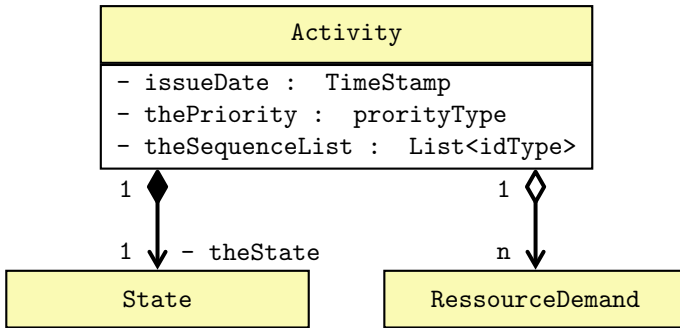


Figure 6.4: Attributes of an Activity, Complementing Figures 5.28 and 6.3

Each instance can also specify an arbitrary number of command sequences to be executed during the activity. Like the Activity itself, sequences and commands can be identified within the schedule via a UUID. Consequently, those sequences are no direct member of the Activity object, but referenced by a list of UUIDs.

Two further attributes playing an elevated role in the scheduling process shall be discussed in more detail. They are the *resource demand* and the *state*. Both are important, because they define the design of the MPT, which will be introduced in the later course of this chapter.

Through the specification of resource demands, the consumption of system resources can be modeled and thus the evolution of the system state over time. Modelling the system state by means of a resource demand enables the detection of conflicts between Activities, as it supports the Activity verification process.

The state indicates to what degree the Activity has already been processed, either on ground or in space. Managing the state is one of the primary tasks of the MPT.

6.2.3 Resource Demand

The major goal of Mission Planning is the creation of a seamless and conflict-free Activity schedule (sometimes referred to as *timeline* too). Conflicts emerge when Activities lead into a system state that would jeopardize the mission, or when they lead into invalid states, which is the case when Activities demand for diametrical system states. To resolve such conflicts, the system state must be predetermined at the planning of new Activities. This can be achieved by means of a resource modelling approach ([128], p. 173–183).

Activity execution is coupled to the availability of system resources. The moment an Activity is executed by the operated system, resources are consumed. A positive consumption describes the case of a resource being used, where a negative consumption describes the mining of a resource. For instance, an Activity that deletes data from memory would mine space for new data and would therefore have a negative memory consumption.

In this context, the system state is defined by the quantity of resources the system provides. System state and resource state are therefore considered as equivalents.

$$\mathbf{R}_{\text{System}}(t) = \begin{pmatrix} R_1(t) \\ R_2(t) \\ \vdots \\ R_n(t) \end{pmatrix} \quad (6.1)$$

Through the planning of Activities, system resources are consumed in a predefined way, which is therefore equivalent to a controlled and predictable alteration of the system state.

The idea of quantifying the system state by means of a resource model has already been applied elsewhere [49, 128]. In some aspects, the approach implemented here follows the concept of a modelling language, which has been applied at German Space Operations Center (GSOC) for timeline generation and conflict resolution [59].

Purpose of such a modelling language is the description of a resource consumption. This requires the definition of system resources, the definition of resource types, the definition of upper and lower bounds, and finally a formalized way to describe the consumption process. The latter is then used for the prediction of the system state.

The consumption process will be described in the following.

6.2.3.1 Resource Level Modelling

The resource consumption of an Activity is quantified by its n resource demand objects (fig. 6.4). These objects indicate how much of a resource is consumed, for how long, and when. The algebraic dependency between the state of a resource (short: *level*), the demand, and the consumption is illustrated in figure 6.5.

The level is a function of the initial state R_0 , a continuous consumption c , and a constant withdrawal b . So, following image 6.5, the level at a time t is then:

$$R(t) = R_0 - b - c(t - t_b) \quad \text{if:} \quad t_b \leq t \leq t_e \quad (6.2)$$

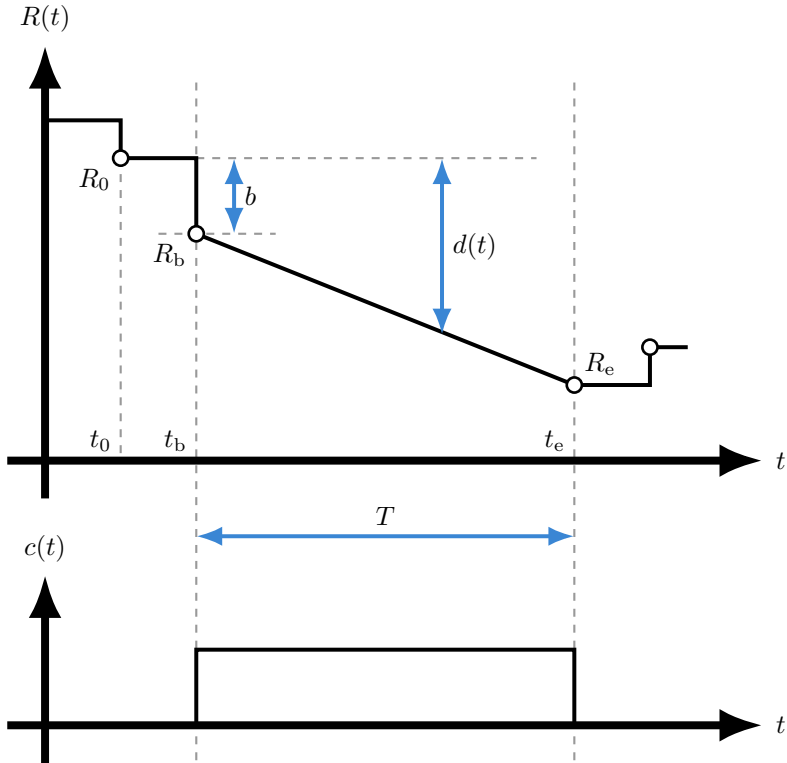
In a more formal way the consumption is quantified by a demand $d(t)$, which is a function of the parameters listed above. The resulting level can then be determined through the integration of that demand function.

$$R(t) = R_0 - d(t) \quad (6.3)$$

$$= R_0 - \int_{t_b}^{t_e} c(t) dt \quad \text{with:} \quad c = \text{const.} \quad (6.4)$$

$$= R_0 - c(t_e - t_b) - b \quad \text{with:} \quad T = t_e - t_b \quad (6.5)$$

$$= R_0 - c \cdot T - b \quad (6.6)$$



- $R(t)$ Resource level over time.
- R_0 Precondition - Last available level prior to execution.
- R_b Level at the begin of consumption.
- R_e Level after the end of consumption.
- T Duration of the consumption
- $c(t)$ Resource consumption $c(t) = -\frac{\partial R}{\partial t}$
- $d(t)$ Resource demand function $d(t) = \int c dt$
- b Initial demand, withdrawn or returned at the beginning.

Figure 6.5: Modelling of the Resource Consumption

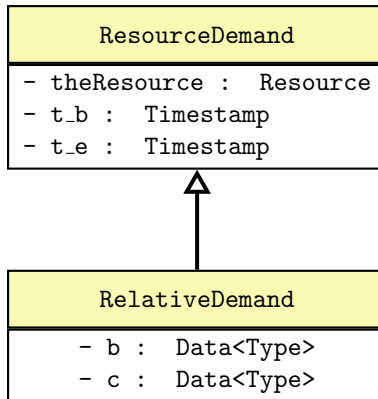


Figure 6.6: Diagram of the Relative Demand Class

A resource of course is not only consumed once, but several times, and during various activities. The resulting resource state at a time t is the sum of all integrated consumptions.

$$R(t) = R_0 - \sum_{i=0}^n d_i(t) = R_0 - \sum_{i=0}^n \left[b_i + \int_{t_{i,b}}^{t_{i,e}} c_i(t) dt \right] \text{ for all } t_{i,*} \leq t \quad (6.7)$$

This formal way of describing a resource demand is referred to as *relative demand*. This means that the required (or the resulting) system state is not specified explicitly, but implicitly through the quantification of a consumption. Accordingly, an Activity can also specify an *absolute demand*. That sort of demand does not describe a consumption, but specifies an explicit condition for a system resource.

The difference between absolute and relative demand with respect to the scheduling process shall be elaborated in the following.

6.2.3.2 Relative Demand

A class diagram of a relative demand object is shown in figure 6.6.

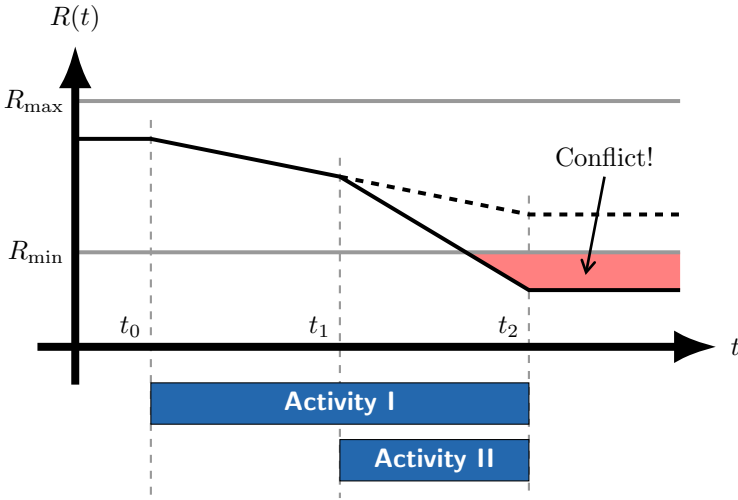


Figure 6.7: Two Activities Excessively Consuming the Same Resource

The relative demand class inherits the generic class *resource demand*. Its attributes are a reference to the consumed resource, and the demand parameters as introduced above: b and c . Furthermore, the object specifies the time frame during which the consumption takes place. A component predicting the state of a system resource through the implementation of equation 6.7 can retrieve all integration parameters from these objects.

A special quality of relative demands is their capability of being accumulated ([59], p. 10). The concept is illustrated in figure 6.7. The moment an Activity starts consuming a resource, the level begins to sink or rise, where

$$\frac{\partial R}{\partial t} = -c(t). \tag{6.8}$$

If another Activity starts using the same resource simultaneously, the consumptions are added, which results in a change of the level gradient.

The use of a resource is limited by the resource's upper and lower bounds R_{\min} and R_{\max} . These need to be specified in the Mission Information Base (MIB) for each resource individually. A conflict occurs, if the consumption of a resource results in a violation of these bounds (fig. 6.7).

Example I

2.5 MB of memory are available at t_0 , when Activity I starts writing data with a rate of 2 MB/min for one minute. If no further Activities are scheduled, 0.5 MB of free space remain after execution at t_2 (fig. 6.7).

If another Activity II starts writing data with the same rate beginning at t_1 , 30 seconds after beginning of Activity I, the total consumption would be 3 MB, which is more than the available memory. Hence, the scheduling of Activity II caused a conflict.

Resource propagation keeping track of the available memory would have detected that conflict through the implementation of the integration schema, as mentioned above.

Example II

Through a narrow definition of R_{\min} and R_{\max} , a relative demand can be used to model the allocation of a resource. For example, a component could be in the state $R_{\min} = 0$ (*unused*), or in the state $R_{\max} = 1$ (*used*). If an Activity is scheduled with a relative demand of $b = -1$ and $c = 0$, the component would be completely allocated. Accordingly, the component can be de-allocated with a relative demand of $b = 1$ at the end of the Activity.

A conflict emerges if two Activities with the same demand $b = -1$ would try to allocate that resource, because

$$R(t) = R_0 - b_1 - b_2 = 2 > R_{\max}.$$

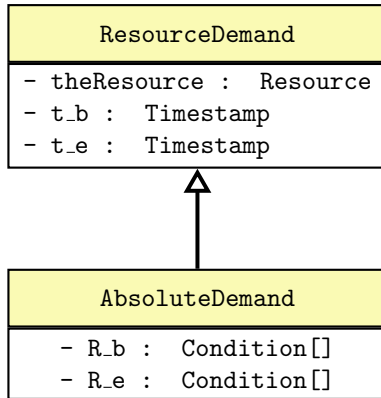


Figure 6.8: Diagram of the Absolute Demand Class

6.2.3.3 Absolute Demand

Similar to a relative demand, an absolute demand describes a transition from one state R_b into a new state R_e (fig. 6.5). The difference is that an absolute demand object explicitly describes the required levels through the specification of one or more conditions, which must be met at the beginning and the end of the consumption. A class diagram is shown in figure 6.8.

The absolute demand inherits the generic resource demand too. Its first attribute is again a reference to the system resource that is of interest. Two sets of conditions describe the required levels at the beginning (t_b) and at the end (t_e) of the consumption.

If no transition is performed, R_b and R_e are identical. This is the case when a system resource just needs to be in a particular state for the execution of an Activity. Four different types of condition can be formulated (fig. 6.9).

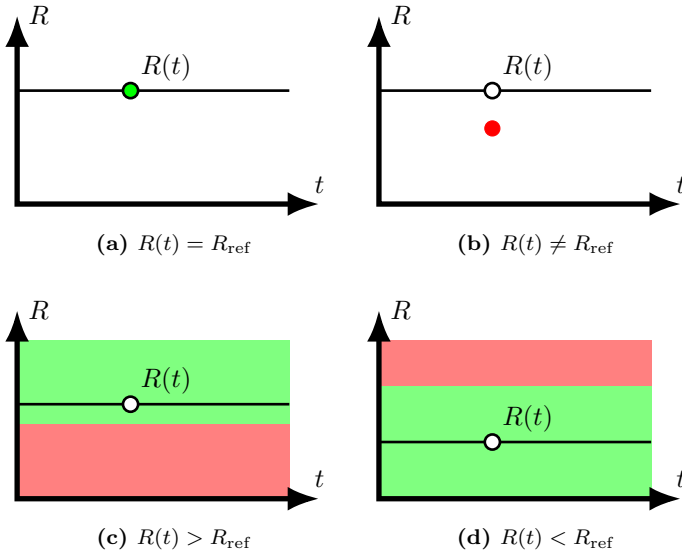


Figure 6.9: Level Conditions that can be Specified by an Absolute Demand

1. The level must be equal to a reference value ($=$)
2. The level must not be equal to a reference value (\neq)
3. The level must be greater than a reference value ($>$)
4. The level must be lower than a reference value ($<$)

If multiple conditions are specified they are linked by a logical OR, which means the demand is met when at least *one* condition is satisfied. This allows for the specification of further expressions like \leq and \geq , or for the specification of different possible options.

An Activity can of course have multiple absolute demands. Conditions from different demand objects are linked by a logical AND, meaning that for a conflict free execution of the Activity *all* demands must be met. For instance, this can be used to narrow down

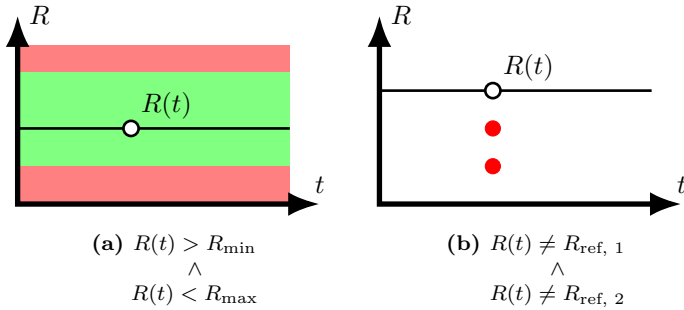


Figure 6.10: Use of Multiple Absolute Demand to Narrow the Allowed Range - For a conflict free execution of the Activity all specified demands must be met.

the allowed range of a system variable (fig. 6.10a), or to exclude particular system states for the Activity (fig. 6.10b).

Example I

For the execution of an Activity, a device temperature must be between 273 and 293 K for the entire duration of $T = t_e - t_b$. Such an Activity would specify two absolute demands: one demand claiming, that the temperature must be greater than 273 K:

$$R_{1,b} = R(t_b) \geq 273K$$

$$R_{1,e} = R(t_e) \geq 273K$$

AND a second demand, claiming that the temperature must be below 293 K the entire time:

$$R_{2,b} = R(t_b) \leq 293K$$

$$R_{2,e} = R(t_e) \leq 293K$$

Example II

A payload Activity changes the system mode parameter from *idle* (0) into *active* (1). The Activity begins at t_0 , and it is assumed that

the mode transition takes 5 seconds. After 30 seconds, the Activity is over and the system mode shall be switched back to *idle* again. Such an activity would specify the following absolute demands:

$$R_{1,b} = R(t_{1,b} = t_0) = 0$$

$$R_{1,e} = R(t_{1,e} = t_0 + 5 s) = 1$$

for the mode transition back into *active*, AND:

$$R_{2,b} = R(t_{2,b} = t_0 + 35 s) = 1$$

$$R_{2,e} = R(t_{2,e} = t_0 + 40 s) = 0$$

for the mode transition into *idle*.

How can Mission Planning benefit from this proposed concept? As mentioned above, system state and the quantity of resource levels are equivalent. This means, the moment an Activity is scheduled, the system state can be predefined. In case of an absolute demand, certain system resources are actively set. Conflicts emerge when various Activities demand for diametrical states of the same resource at the same time.

Depending on the considered system parameter, a resource state is only valid for a limited time frame. For example, a battery state of charge information is only valid for a couple of minutes. After that the value needs to be updated. Thus, a conflict check can only be performed if recent satellite TM is available, or if the value is continuously predicted.

Change of system parameters which don't become invalid after a certain amount of time, (e.g. system mode), can be scheduled and checked easily without considering the timestamp of the last telemetry data point, or the timestamp of the last scheduled change. This is because those parameters only change when actively set. An unscheduled change of a permanently valid system parameter usually

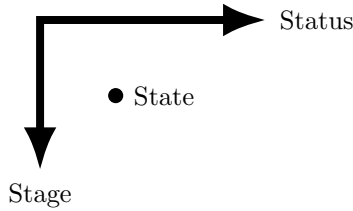


Figure 6.11: Activity State - During its *life*, an Activity passes a series of states. The state is modeled by means of a two-dimensional approach. The *stage* indicates to what degree the Activity has been processed (in space or on ground). The *status* indicates, whether the Activity processing is currently active, suspended or terminated.

indicates a system failure, which can be intercepted by the Mission Planning through this approach.

6.2.4 State

Before an Activity is executed by the spacecraft, it needs to pass a series of processing stages. At first, it somehow needs to be inserted into the system schedule on ground, after that it is uplinked to the spacecraft, inserted into the on-board schedule, or executed instantaneously. In a last step, that execution is verified by means of the satellite TM, before the Activity is *closed* eventually.

Managing an Activity on ground requires various processing steps during each of these stages. For instance, a recently requested Activity is handled fundamentally different from an Activity currently executed by the spacecraft.

For a proper Activity handling on ground, the state of the Activity must be resolved precisely, so that the correct processing mechanisms can be applied. A two-dimensional approach has been selected for that purpose, where the state of the Activity is represented by a vector (fig. 6.11). Its vertical component, the *stage*, indicates the degree to which the Activity has been processed. This covers ground processing as well as the processing by the spacecraft. The

horizontal component, the *status*, indicates whether the Activity is still being processed or not. The four possible statuses are:

- **In Progress** - This is the nominal status. If the Activity is in progress, it is being processed until it is closed eventually.
- **Suspended** - Under certain circumstances, the handling of the Activity can be suspended. A suspended activity won't be processed neither on ground nor in space until it is being resumed again.
- **Failed** - Activities enter the status failed in case something goes wrong or the Activity is rejected. Failed activities cannot be resumed anymore and will be closed.
- **Closed** - This is the final status each Activity will enter eventually. Closed Activities cannot be opened again.

Purpose of the MPT is incrementally altering the stage of an Activity. The state diagram of such an increment is shown in figure 6.12 for an arbitrary stage n .

As long as the status is *in progress*, and all preconditions are satisfied (e.g.: All commands have been successfully uplinked!), the Activity can be altered towards the next stage ($a \rightarrow b$). This process is repeated until the Activity reaches its final stage n_{final} , after which it is closed eventually ($c \rightarrow d$).

If the preconditions for a nominal stage alteration are not or cannot be satisfied, the status is set to *failed*. Setting an Activity into the failed status is always accompanied by an alteration of the stage ($a \rightarrow f$). This shall indicate that reaching the next stage $n + 1$ has failed.

Under certain circumstances (e.g. in case of a conflict) it can be necessary to suspend the Activity processing. A suspended Activity cannot be altered, unless it is resumed again ($e \rightarrow a$). Altering a suspended Activity means that the Activity has failed ($e \rightarrow f$).

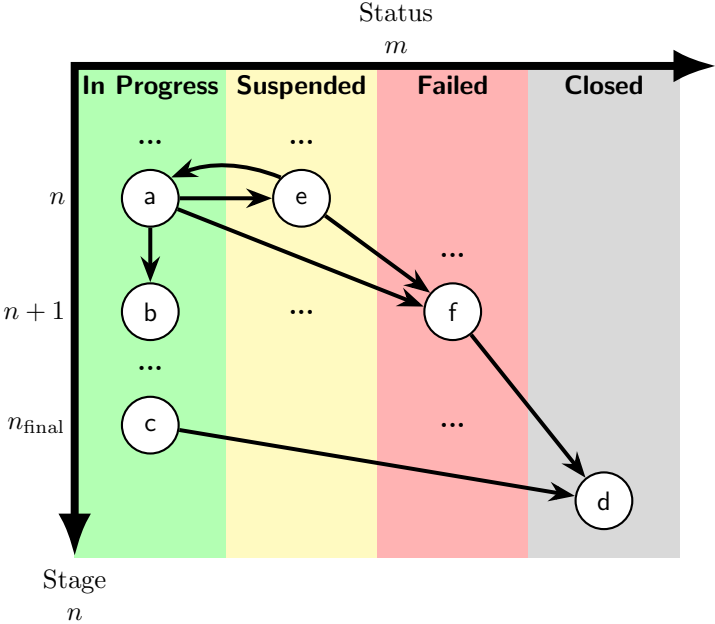


Figure 6.12: Incremental Change of the Activity State

This happens for instance when the execution time of a suspended Activity has elapsed.

The increment as sketched in figure 6.12 is repeated a couple of times until the Activity has reached its final stage or has failed. The complete state diagram is shown in figure 6.13. Incrementally altering the Activity state in the described fashion has the benefit that the Activity state can be set with a very limited number of simple functions (fig. 6.14). Since the next state is always a function of the existing one, and since only a limited number of basic transitions are possible (fig. 6.12), those functions don't need any arguments, which protects the Activity state from being corrupted.

Figure 6.13 depicts the entire state diagram an Activity goes through during its *life*. It begins when the Activity is *requested* by the initiator. If it is not rejected, the Activity is marked as *scheduled on ground*. During this state the Activity is prepared for transmission, which means that command release times are determined and the activity is constantly checked for conflicts.

Upon release time, the Activity, respectively the commands in the sequences, are transmitted to the spacecraft and added to the on-board schedule, if not executed right away. Once the on-board software has started executing the commands, the stage *in process* is reached.

An executed Activity of course cannot be suspended anymore. After execution, the Activity is verified by means of satellite TM, and closed eventually.

In order to enable a human operator to follow the current Activity state conveniently, each of the 19 states can be interpreted by means of a translation table, which is implemented in the state object (tab. 6.2).

6.2.4.1 State Machine

The conditions for an Activity to enter the next stage strongly depends on the present status of the Activity. If for example the Ac-

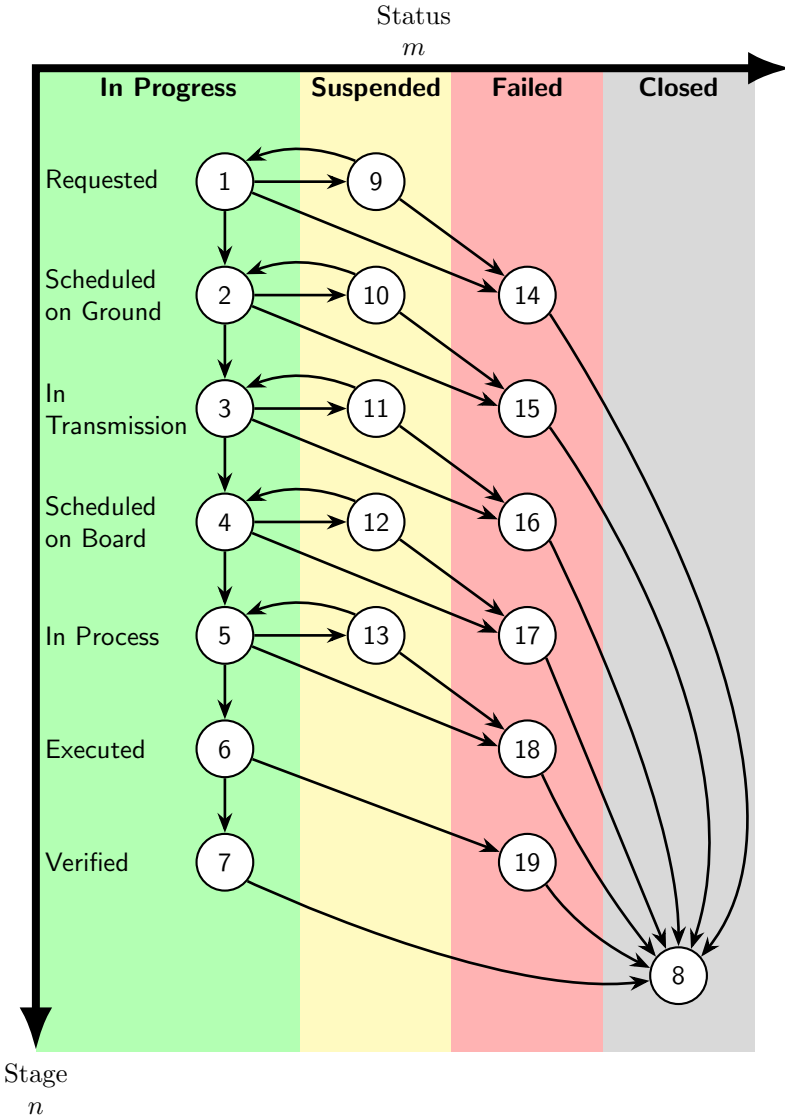


Figure 6.13: Complete Activity State Diagram

State
+ alterStage() : state
+ susepend() : state
+ resume() : state
+ setFailed() : state
+ close() : state

Figure 6.14: Selection of Member Functions used for the State Modification - Each function returns the resulting two-dimensional state vector, informing the caller about the effect of the function call.

Table 6.2: State Translation Table - Each state can be translated into a human understandable interpretation.

State	Translation	Status
1	requested (initial state)	
2	scheduled on ground	
3	in transmission	
4	scheduled on board (transmitted)	In Progress
5	in process	
6	executed	
7	verified	
8	closed	Closed
9	on hold	
10	withdrawn	
11	transmission suspended	Suspended
12	suspended on board	
13	execution suspended	
14	ground scheduling failed (rejected)	
15	release failed	
16	transmission failed	
17	processing failed	Failed
18	execution failed	
19	verification failed	

tivity is *scheduled on ground* (2), the next stage *in transmission* is entered when the release time of the first command is reached. Switching from *in process* (5) to *executed* is possible the moment the Mission Control System (MCS) has verified the execution of all commands. Purpose of the MPT is to implement the incremental state machines for each of the defined states (1–19).

If the Activity features a command sequence and/or a resource demand (sec. 6.2.3.1), the Activity state can be derived from the state of these commands and from whether the spacecraft state satisfies the resource demand or not. If the Activity neither specifies a command sequence nor a resource demand, the state of the activity depends on the state of the child objects.

Due to the simplicity of the state increment as shown in figure 6.12, the parent *stage* can be determined by means of some simple functions.

As long as the Activity has not entered the stage *in process* ($n = 5$), the stage of the parent activity is equal to the maximum child¹ stage.

$$n_{\text{parent}} = \max\{n_{\text{child}, 1}, n_{\text{child}, 2}, \dots\} \quad \text{if } n_{\text{parent}} < 5 \quad (6.9)$$

For instance, the parent stage becomes *scheduled on board* if that is the stage of at least one child even though others are still *in transmission*. The moment the stage *in process* is reached, the parent stage is equal to the minimum child stage.

$$n_{\text{parent}} = \min\{n_{\text{child}, 1}, n_{\text{child}, 2}, \dots\} \quad \text{if } n_{\text{parent}} \geq 5 \quad (6.10)$$

So the parent is only considered *executed* if all child tasks have been executed, as well as the parent is only considered *verified* if all child tasks have been verified.

¹The concept of parent and child relationship between Activities is introduced in section 6.2.5

The determination of the parent *status m* is a little more sophisticated. The following logic is foreseen:

- The parent status is *in progress* if at least one child is in progress and no child has failed.
- The parent status is *suspended* if all children are suspended.
- The parent status is *failed* if at least one child has failed.
- The parent status is *closed* if all children are closed.

Unlike the stage, the status can also be set top-down. Suspending the parent activity triggers a suspension of all child activities, as resuming the parent triggers the attempt of resuming the children. Accordingly, all children are set failed if the parent is set so.

6.2.4.2 On-Board Schedule Synchronization

The execution of time-tagged TCs is usually managed by one or more schedules within the satellite on-board software ([9], p. 104). Under certain circumstances, e.g. in case of a system mode fallback, these schedules usually become suspended. Activities can be associated with these on-board schedules. Through the creation of an Activity schedule for a particular on-board schedule, and through monitoring the appropriate system parameter indicating the run-rate of the on-board schedule, the Activity status can be synchronized with that on-board schedule. This allows for the automatic suspension of all affected Activities in case of a system mode fallback. After a successful recovery and a resume of the appropriate on-board schedule, these Activities are resumed too.

The implementation of a synchronization with an on-board schedule is not further pursued here. However, the possibility of creating independent Activity Schedules for various system components (such as on-board schedules) will be dealt with in section 6.5.1.2, in the course of the discussion of the constellation planning.

6.2.5 Nested Activities

An important quality of an Activity as a scheduled object is the possibility of nesting it. This allows for the splitting of a certain task (parent) into various subtasks (children) as shown in figure 6.15a. The implemented concept is recursive, which means that a child of a parent can have further children and become a parent itself.

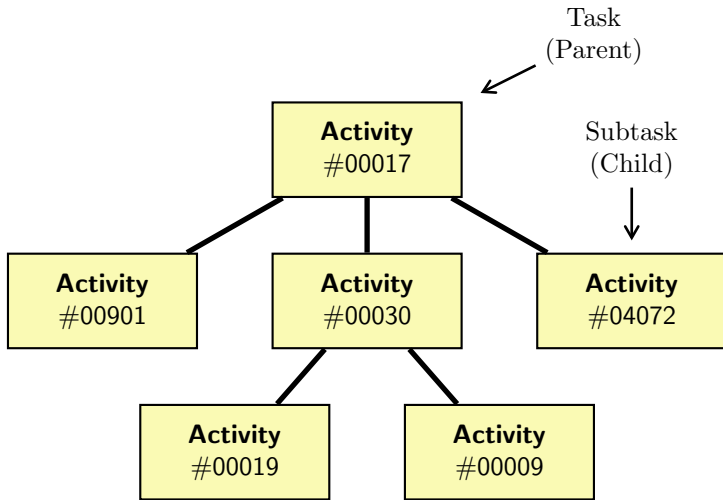
A child Activity is no owned attribute of its parent. Instead, parent and child objects are referenced by means of their UUID (fig. 6.15b). This is because each Activity shall be managed separately as an independent element in the schedule. Yet, a software component like the MPT that manages an Activity must be capable of accessing those referenced objects. This can be achieved efficiently by means of pointer maps as implemented by [98]. A pointer map is a concept that allows for call by reference accesses to objects in the memory. The object pointers in the map are identified with a unique key, in this case the Activity UUIDs.

The concept does not make a restriction in terms of the Activity executor, and thus the schedule the Activity is part of. It is foreseen that parent and child tasks can be executed by different entities, as well as each child can have a different executor. As a consequence, each child Activity under a parent task might be executed by a different system and thus be managed by a different MPT.

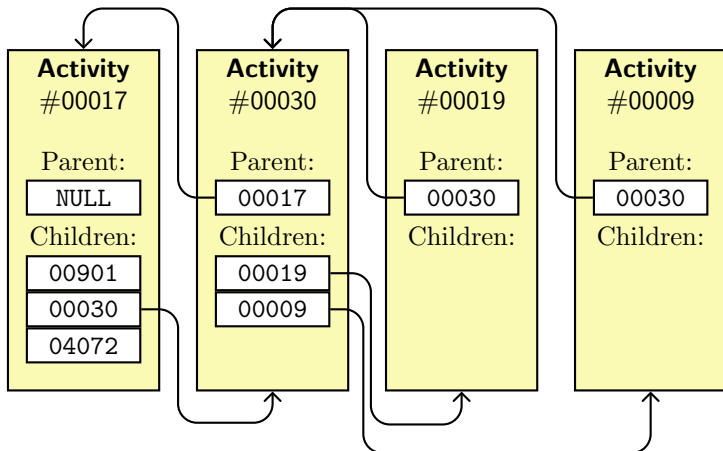
Examples where this concept is taken advantage of are constellation Activities, discussed in section 6.5.1, and *Link Activities*, discussed in section 6.2.6.1.

6.2.6 Derived Activities (Examples)

In satellite operations there are always types of activities which play an elevated role to the scheduling process and/or the conduction of the mission. Two of such are discussed in the following.



(a) Nested Structure of Activities - Parents can have a number of children, as those children can be parent of their children.



(b) Referencing Nested Activities - Children are no attributes of the parent activity, but referenced by means of their ID.

Figure 6.15: Structuring and Referencing Nested Activities

6.2.6.1 Link Activity

A *link* describes a remote data connection between at least two communication partners. Physical links between ground and space are usually realized by means of electromagnetic waves, where inter-satellite links (ISL) often rely on optical systems. Due to the larger bandwidths and the lower regulatory efforts, optical systems are recently considered for space-to-ground communication too [78].

A temporary link between a satellite and a ground station is also referred to as *pass* ([134], p. 180).

Successfully enabling a data link, requires certain activities by all communication partners. Prior to communication, receiver and transmitter must be switched on, and the receiver must be locked on the sender signal, which requires a line of sight connection between both parties, and synchronization of the receiver with the carrier wave. In turn, communication is terminated if at least one component in the link chain is disabled, or if the line of sight connection is lost.

For the scheduling process within the MPT, the link planning plays an important role. Due to the nature of space flight, communication with a satellite is usually limited to a couple of minutes a day. Consequently, satellite activities must be uplinked in advance and the satellite takes care of the time-tagged execution. The question is: When can those commands be uplinked? The *Link Activity* is the means of determining when and how many TCs can be released.

A Link Activity describes a single direction communication; either an uplink, or a downlink. If a pass shall be used for up- and downlink, two Link Activities must be scheduled. Each Link Activity is parent of a couple of child Activities. Figure 6.16 shows the example structure of a Link Activity between a satellite and a Ground Station (GS). The child Activities comprise those technical processes in space and on ground which are necessary to enable

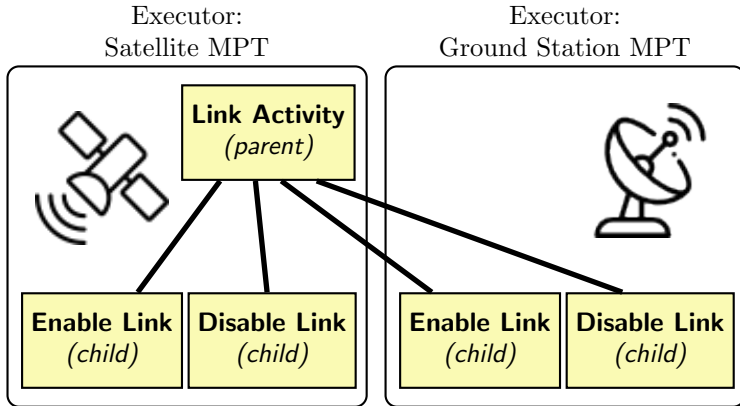


Figure 6.16: Link Activity - A link Activity is composed from a couple of child activities, which enable the link and disable the link eventually. Enabling/Disabling a link requires action by both communicating systems (e.g. the satellite and a ground station). Therefore, the appropriate child activities need to be planned in the satellite schedule as well as in the ground station schedule.

the link (e.g. transmitter activation), and those to disable it again. Consequently, these child Activities have to be added to the satellite schedule as well as the ground station schedule².

The time frame during which commands can be released is the duration between acquisition of signal (AOS) and loss of signal (LOS). AOS is reached if all link enabling Activities have been executed successfully and the phase has begun. The phase is the period during which GS and satellite have a line of sight connection (sec. 5.3.2.4). In turn, LOS is reached if either the phase is over or one system has started executing a link disabling Activity. An example schedule for a ground station pass is shown in figure 6.17.

By means of the link duration T , the available bit rate, and the frame size the number of commands that can be uplinked during one

²Through the specification of appropriate resource demands by the GS activities, the GS can be allocated, and thus blocked for other satellites. See Example II in section 6.2.3.2.

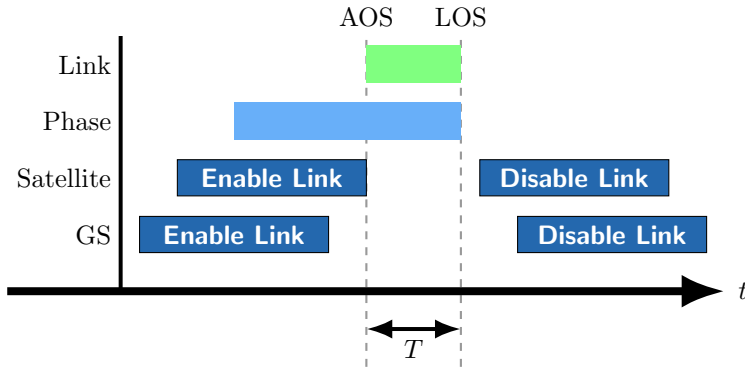


Figure 6.17: Link Duration - The time between AOS and LOS during which data transmission is possible.

pass can be determined. The principle also applies to the scheduling of a TM downlink.

In figures 6.16 and 6.17 an example of a ground station pass is considered, which describes a link between two communication partners. The concept however does not make a restriction in terms of how many systems are involved into the transmission. If for example relay systems shall be used, the appropriate amount of child Activities can be added to the link Activity. In that case, AOS is reached if the link is enabled successfully by all systems in the communication chain. The conditions for LOS are accordingly.

6.2.6.2 Maneuver Activity

A maneuver describes a change of the satellite orbit, achieved through a deliberate acceleration or deceleration along the flight path ([91], p. 117). In space flight, maneuvers are used

- for the spacecraft injection into a target orbit,
- for rendezvous,
- for docking,

- for de-orbiting and landing,
- or for station-keeping.

Station-keeping is required to encounter for orbital drifts evoked by natural disturbances [17]. Formation satellites in particular demand for a steady station-keeping to maintain a target relative position between each other [129].

The change of the orbit can be achieved through the utilization of external impacts (swing-bys, atmospheric drag maneuvers, etc.), or through the active appliance of thrust. In case of an impulse maneuver, a relatively large amount of thrust is applied over a short interval, while low thrust maneuvers slowly change the orbit over a longer period [91]. All of these possibilities are covered by the concept described here, since the detail principle based on which the orbit change is achieved is not taken into account by the Mission Planning.

A *Maneuver Activity* is characterized by the initial orbit, the target orbit, and a duration. An orbit can be described e.g. by the six Keplerian elements. So, within the Maneuver Activity initial and target orbit are described through the appropriate number of absolute demands specifying the states of these six elements at the beginning (t_b) and the end (t_e) of the maneuver:

$$\mathbf{R}_b = \begin{pmatrix} a(t_b) \\ \epsilon(t_b) \\ \iota(t_b) \\ \Omega(t_b) \\ \omega(t_b) \\ \tau(t_b) \end{pmatrix} \quad \text{and} \quad \mathbf{R}_e = \begin{pmatrix} a(t_e) \\ \epsilon(t_e) \\ \iota(t_e) \\ \Omega(t_e) \\ \omega(t_e) \\ \tau(t_e) \end{pmatrix}$$

A Maneuver Activity can be verified through comparison of the planned target orbit (\mathbf{R}_e) with the one that has actually been achieved. Accordingly, the scheduled initial orbit (\mathbf{R}_b) can be compared with the real initial orbit in order to estimate the success of the maneu-

ver in advance. A noticeable orbit deviation at the beginning of the activity would trigger as suspension of the Maneuver.

Another resource that is consumed during a maneuver is the propellant mass provided by the spacecraft. Specifying the fuel consumption allows verifying whether the spacecraft can provide enough specific impulse for the maneuver.

The determination of the maneuver parameters: the orbits, and the required impulse is not implemented within the MPT. This must be achieved by appropriate Agents or the Flight Dynamics Tool (FDT). Discussing them is out of the scope of this work.

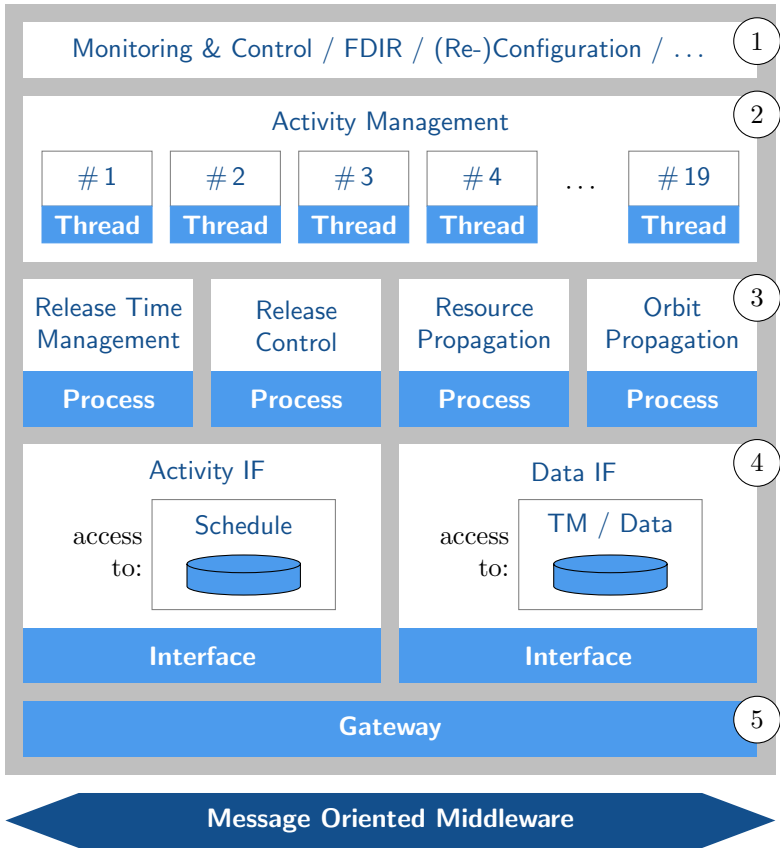
6.3 Mission Planning Tool Architecture

Figure 6.18 features an overview of the functional MPT architecture. It is the implementation of a layered approach, where each layer is either responsible for a certain aspect in the scheduling process, or for the maintenance of the MPT itself. The layers are hierarchically organized in a top-down fashion, where the upper layers either have the authority on the lower ones, or define their work load. An introduction of each layer is pursued in the following.

6.3.1 Configuration Layer

The configuration layer is the top layer of the MPT architecture, as it is basically in charge of all processes executed by the layers underneath.

The MPT implements multiple threads and processes for the various purposes. At component launch, the configuration layer takes care that each process and each thread is engaged, and that all elements within the tool are initialized as specified by the System Information Base (SIB) (sec. 5.3.3.1).



- ① Configuration Layer (sec. 6.3.1)
- ② Scheduling Layer (sec. 6.3.2)
- ③ Operative Layer (sec. 6.3.3)
- ④ Interface Layer (sec. 6.3.4)
- ⑤ Communication Layer (sec. 6.3.5)

Figure 6.18: Mission Planning Tool Functional Architecture

After the tool has been launched, the configuration layer is in charge of monitoring all the other software elements within the component. That monitoring & control feature allows tracking the state of each thread and process, as it allows for starting, stopping, pausing, or resuming them. Purpose of this feature is enabling a user or an internal Failure Detection Isolation and Recovery (FDIR) to control and re-instantiate threads and processes if necessary.

6.3.2 Scheduling Layer

In section 6.2.4 of this chapter it has been pointed out that an Activity can be in 19 different states (fig. 6.13). These states precisely define what happens with an Activity during the scheduling process. Purpose of the scheduling layer is the implementation of state machines that determine the current state of an Activity based on present boundary conditions. That state is then the basis for the actions happening on the layers underneath. These actions then have an impact on the boundary conditions, which then are considered for the update of the state again. That process continues until the Activity has passed all stages and is closed eventually.

For the determination of an Activity state, the scheduling layer considers a number of boundary conditions. If an Activity contains a command sequence and/or specifies a resource demand, the Activity state is determined on the basis of:

- the state of the operated system, respectively the state of the consumed resource(s),
- or the state of the commands in the sequence

If an Activity is parent of one or more child Activities instead, the state of the parent is determined only on the basis of the child Activities' states. At this point of the MMOS development it is not allowed that Activities have children, *and* have command sequences and/or resource demands. This is simply due to the fact that consid-



Figure 6.19: MPT scheduling layer

ering children *and* commands/resources for the state determination could give ambiguous results.

The determination of the parent state as function of the child states has been introduced in section 6.2.4.1 and won't be discussed further in this section.

The state determination on the basis of commands and the resource demands is a little more complicated though. It also depends on the present state of the Activity too. This is why the MPT scheduling layer implements 19 independent threads for this task (fig. 6.19). At each state an individual thread checks those boundary conditions, which must be checked for an alteration of the Activity in that particular state. If a condition is satisfied, the Activity is transferred into the next stage (fig. 6.13). In case of a conflict or a problem, the Activity is either *suspended* or marked as *failed*.

Each thread implements a state machine like the one depicted in figure 6.20. That figure displays those checks, which must be performed during state 1 (*requested*). As a result of these checks, the Activity is either *suspended*, or altered towards the next stage: *scheduled on ground*.

Almost every loop begins with a resource check. If that check indicates a conflict that can be resolved by suspending one or more Activities with lower priority, or if no conflict is detected at all, the Activity processing is kept on. Conflicts that cannot be resolved result in the suspension of the Activity.

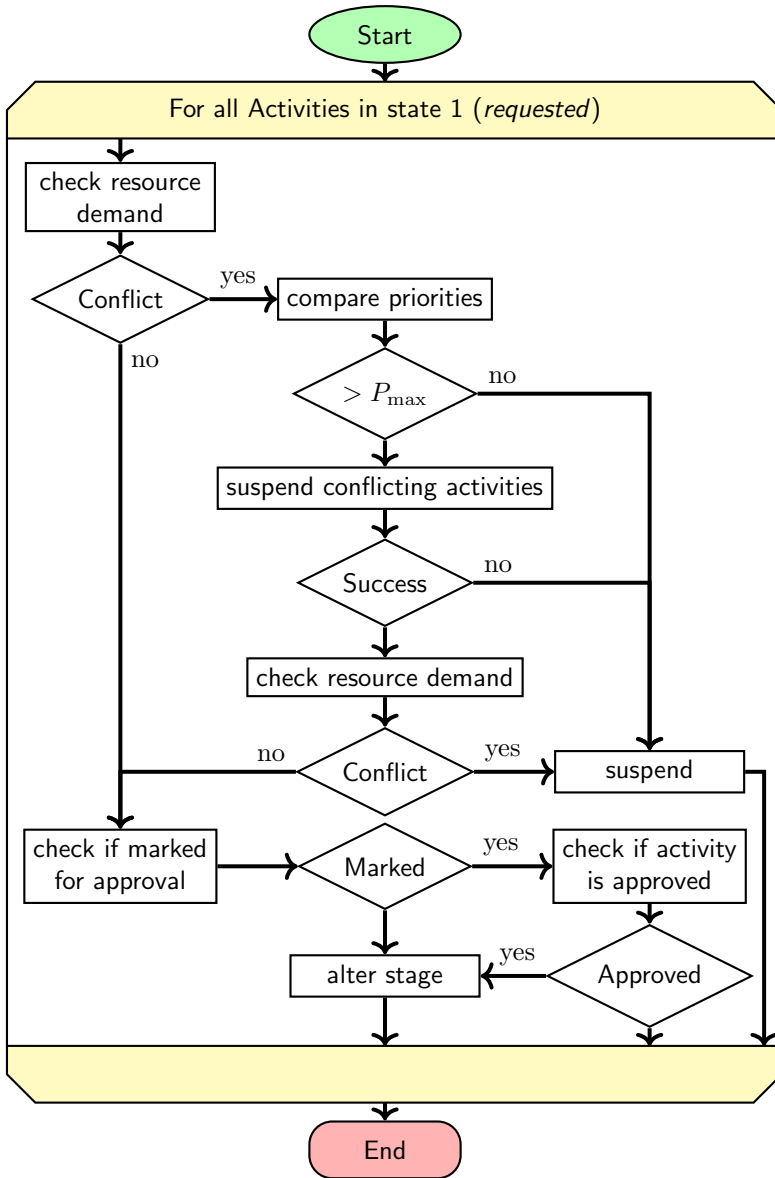


Figure 6.20: Simplified Flow Diagram of a Scheduling Thread - The flow diagram shows the transition of a *requested* Activity into the next stage.



Figure 6.21: MPT operative layer

Suspended activities are handled by threads 9 to 13. These threads constantly check if the conditions for the resume of an Activity are satisfied. If a resume is not possible anymore, for instance when the execution time has passed, the Activity is marked as *failed* and is *closed* eventually.

All state machines, which are implemented in threads 1 to 19, are compiled in appendix B.2.

6.3.3 Operative Layer

During each state of the scheduling process, particular actions need to be performed by the MPT. For example, when an Activity is in state 3 (*in transmission*), the commands need to be released. And when an Activity is *in progress*, its recourse demand must be considered for the detection of conflicts. The operative layer (fig. 6.21) is the layer where these actions are implemented. It consists of the following components:

- the **Release Time Management**,
- the **Release Control**,
- the **Resource Propagation**,
- and the **Orbit Propagation**.

6.3.3.1 Release Time Management

Purpose of the release time management is the determination of the command release times, which refer to the time when the command is sent from the schedule to the executing system via the MCS. For the determination of that time, the release time management must consider a number of boundary conditions though, such as

- the command execution times,
- the duration of the link, respectively the times of AOS and LOS,
- the available uplink bandwidth,
- and the estimated size of the TCs.

Before a release time can be determined, an appropriate window for the uplink must be identified. That window must be before the command execution time. Windows for command uplink are the link durations between AOS and LOS of a satellite pass, which can be determined on the basis of the scheduled *Link Activities* (sec. 6.2.6.1), and the *phases* during which satellite and ground station have a line-of-sight connection (fig. 6.17). These phases, and the duration of those phases can be requested at the FDT via the appropriate interface (sec. 5.3.2.4).

The command release time is determined automatically for all commands in Activities, which are in state 2 *scheduled on ground* or in state 11 *transmission suspended*.

This is the nominal process, which is quite straight forward. The determination of the release times can become rather complicated though, especially in case something unforeseen happens. For instance, the command transmission can fail. In this case the release time management must trigger another attempt through the definition of a new release time. Or the duration of a link can be too short, or the available bandwidth is too low to uplink everything, such that the upload of Activities must be split among different passes.

Situations like these must be covered by the release time management to guarantee a reliable transmission to the spacecraft.

6.3.3.2 Release Control

The working principle of release control is very simple. Its only purpose is to mark those commands, which are ready for release.

As mentioned in section 5.3.2.1, commands are released directly from the schedule, and transmitted to the MCS via the command interface. The release control is the component of the MPT which triggers the transmission by tagging those commands in the schedule, which shall be released.

Release control tags all commands of all Activities, which are in state 3 *in transmission*. This enables transmit by the schedule, when release time is reached.

Accordingly, all tagged commands of Activities, which are *not* in stage 3 will be untagged. This way the transmit by the schedule is disabled again.

6.3.3.3 Resource Propagation

In section 6.2.3 the resource demand attribute of an Activity has been introduced. The parameters within these objects can be used to propagate the state of a resource variable, through the application of an integration scheme. The resource propagation is the component where this integration takes place.

The propagation of a resource variable into the future requires an initial state $R(t)$, which is the last known state of the resource gained from telemetry, and a number of boundary conditions. These boundary conditions are the integration parameters from the resource demand objects (fig. 6.22). With these information in place the run of a resource variable can be simulated.

The propagation of a resource is triggered asynchronously every moment the initial state is updated, or when an Activity consuming

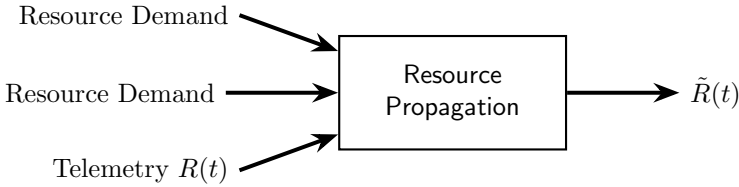


Figure 6.22: Resource Propagation Black Box Functionality - The estimated resource level $\tilde{R}(t)$ is a function of the demands and the actual level $R(t)$, gained from telemetry.

that resource is suspended, resumed, removed from, or added to the schedule. The simulation results are stored in a propagation database for each resource separately. A conflict emerges, when the simulation shows that a resource is consumed extensively, or when diametrical demands cannot be satisfied.

Furthermore, the resource propagation keeps track of all the Activities which will consume a resource. In case of a conflict, that information is requested by the threads in the scheduling layer. Resolving a conflict means, that at least one of those Activities involved needs to be suspended. That decision is made in the scheduling layer as well. Suspending one Activity automatically triggers the propagation process again. The conflict is resolved, if after the suspension the simulated resource level remains within limits, and no diametrical demands are detected.

6.3.3.4 Orbit Propagation

The concept of orbit propagation is very similar to the concept of resource propagation. The major difference is that the orbit propagation does not simulate system resources, but the future satellite orbits based on the scheduled maneuver Activities (sec. 6.2.6.2).

In case of the orbit propagation the initial state $R(t)$ is the present orbit. The respective orbital parameters can be derived from telemetry or gained from other sources like TLEs. According to the resource propagation, the orbit propagation is triggered

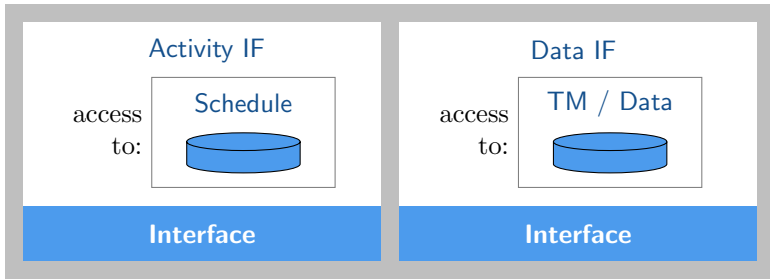


Figure 6.23: MPT interface layer

every moment the present orbit information has been updated, or when maneuver Activities are suspended, resumed, removed from, or added to the schedule.

Like the resource propagation, the orbit propagation keeps track of planned maneuver activities. A conflict emerges, when the specified target orbits cannot be reached. In such a case the same mechanisms for conflict resolution apply as described in the previous section.

Unlike the resource propagation, which can be achieved through very simple integration schemas (sec. 6.2.3.1), the simulation of the future orbit requires specialized program libraries. This is why the orbit prediction does not happen internally the MPT, but within the FDT, which has been described in section 5.2.2.3.

6.3.4 Interface Layer

As indicated many times, the MPT does not keep any information, it only reads and updates information that is stored elsewhere. The interface layer is the means of getting access to that data.

Three different types of interfaces implemented in this layer are:

- the **Activity** interface,
- the **Data** interface,
- and the **Phase** interface (not shown in figure 6.23).

The Activity interface is the object that enables access to the system schedules. All read and write requests by the scheduling layer components or the operative layer components involving schedule data are made via the Activity interface. This covers:

- reading and updating Activity states,
- reading command statuses,
- updating release times,
- tagging commands for release,
- etc.

The data interface allows accessing the satellite TM, managed and provided by the Data and Archives System (DAS) (sec. 5.2.2.2). This information is needed for the determination of the beginning conditions $R(t)$ of the resource and orbit propagations. During the Activity verification process carried out by thread number 6, the interface is needed to compare the actual resource state with the propagated one.

Via the phase interface the MPT can request the occurrences and the duration of satellite passes over ground stations at the FDT. These information are needed for the determination of release times by the release time management.

6.3.5 Communication Layer

All data traffic between the MMOS components is routed via the MOM. Responsible for the connection of the MPT to the MOM is the communication layer. The software component that facilitates the communication is the so called *gateway* (fig. 6.18). By means of the gateway, the MPT can subscribe for the different message queues, which handle and buffer the messages between the components.

Through the creation of different message queues in the middle-ware, and the definition of message types, the gateway ensures that

information is addressed to the correct interface. In turn, outgoing messages are automatically added to a particular sending queue. The MOM ensures that messages are transferred to the target component, and that messages are not getting lost in the event of the receiving end being down at the moment of transmission. For the latter, the MOM keeps messages in the buffer until the transmission has been executed.

6.4 The Schedule

To this point of the discussion, it always has been pointed out that the MPT manages the mission schedule, and that it has the sovereign authority of it. What has not been mentioned yet is that the schedule itself is a separate, self-contained component within the MMOS, which fulfills a series of dedicated functionalities. MPT and schedule are the two sides of the same medal though, which is why a description of the MPS could not be sufficient without going into some details of the schedule architecture too.

The functional architecture of the schedule is depicted in figure 6.24. In a way it is similar to the design of the MPT, which is due to the fact that all components integrated into the MMOS must feature the same communication layer.

On top of the gateway, the schedule implements the counter part of the Activity interface by means of which the schedule receives the requests from the MPT (or any other component demanding schedule information). The requests are handed over to a component called the *Schedule Management*, which handles the requests and returns the required information. Purpose of that schedule management is not only the handling of such request, but also the maintenance of the database underneath.

The schedule data is persistently stored in a PostgreSQL database [127]. Whenever a request is received, the schedule management converts that request into a query statement in order to return the

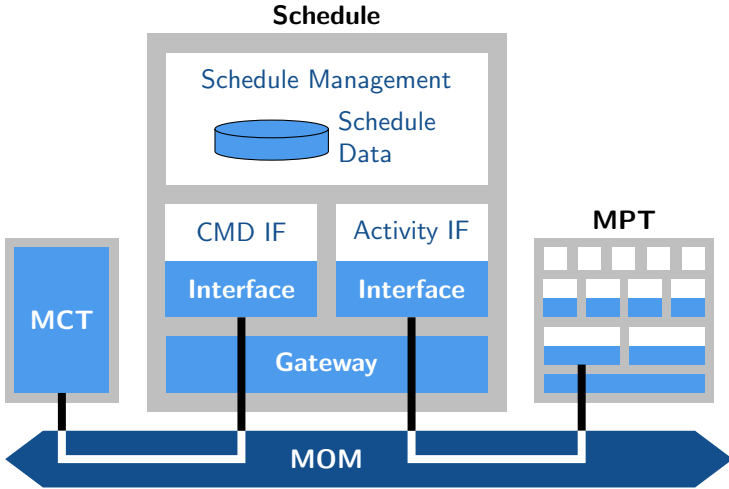


Figure 6.24: The System Schedule Integrated Into the MMOS Architecture

requested data or to write an entry. Since the schedule management is the instance actually writing the low-level schedule data, it is also the component in charge of the schedule consistency. Thus, it must protect the schedule from unauthorized access, as it must prevent the schedule from being corrupted (e.g., through simultaneous access).

As mentioned during the discussion of the MPT release management, the schedule is also the component that transmits the commands to the MCT. Releasing commands from the schedule is just a matter of querying the commands, which are tagged for release (sec. 6.3.3.2). When release time is reached, commands which are appropriately tagged are sent via the command interface. This is also the interface by means of which the schedule receives acknowledgement information back from the MCT. These acknowledgement information is written into the schedule by the schedule management and can be requested later on by the other components via the Activity interface.

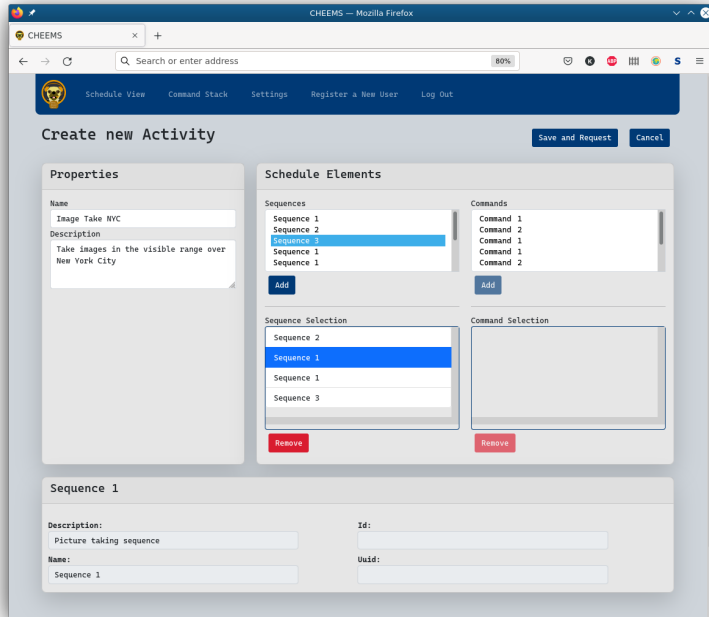


Figure 6.25: Activity Editor of the MPS GUI, implemented by [5]

6.4.1 Graphical User Interface

The MMOS Graphical User Interface (GUI) is realized as a web-application. Through the implementation of the interfaces introduced in section 5.3.2 that application allows interacting with the various MMOS microservices. Of course, this applies to the MPS and the schedule component too. Via the Activity interface implemented by the GUI, a user can interact with the system schedule.

If a mission planner has write access to the schedule, the application allows to create new Activities and add them to the timeline. Existing Activities can be edited. That write access is very limited though. For example, users are neither allowed to manually alter

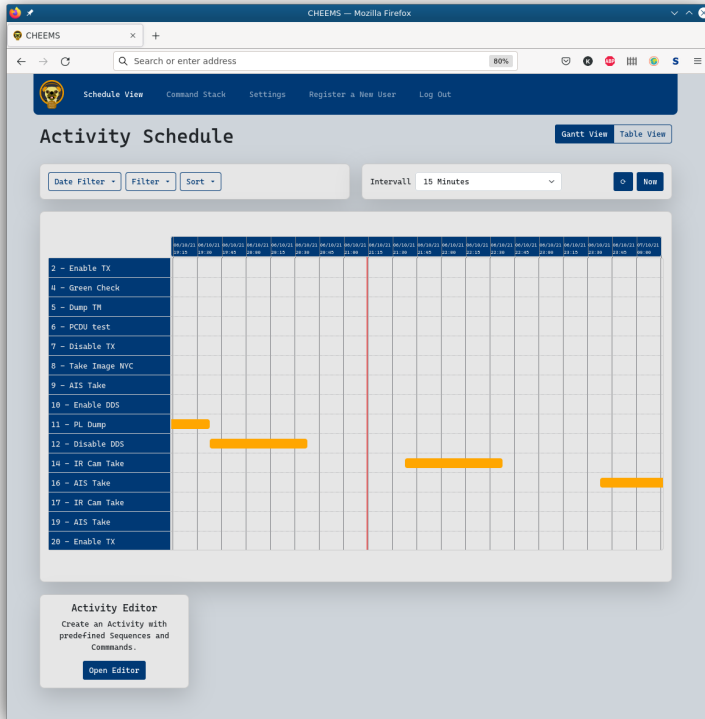


Figure 6.26: Timeline View of the MPS GUI, implemented by [5]

the Activity state, because that is entirely done by the MPT, nor can they maintain the database underneath.

An example of a mission timeline as displayed within the GUI is shown in figure 6.26. In this view, the planned activities are displayed in a Gantt chart that shows the operator the progress of the mission.

The command list derived from that timeline is displayed in a different view. In that command view, the operator can also create and add single commands to the schedule. During automatic operations this is not recommended though, because the system state

resulting from the execution of single commands is not considered by the resource propagation, and can cause conflicts.

The first version of the MPS including MPT, schedule, and this GUI was implemented by [5].

6.5 Constellation Planning

This section shall recap what has been introduced so far and show how the proposed concept supports constellation operations.

Proposed has been an architecture for a configurable and extendable MMOS, supporting automation by closed loop operations. Through the definition of appropriate interfaces in space and on ground, the communication between MMOS and operated system has been broken down into several abstraction layers. Those layers are reflected in the control process and thus by the MMOS design.

The MCS interfaces the operated system on application layer, as it addresses the processes/services implemented in the satellite on-board software. A state of the art protocol for that application layer communication is the Packet Utilization Standard (PUS) [41]. Such an application layer protocol does not yet support higher-level system operations and scheduling though, which is why a protocol agnostic system layer interface has been introduced, that is the *Activity*. The Activity as a top-level abstraction layer, supports the scheduling of interrelated system level processes and the resolution of conflicts even across system boundaries.

With all of that being settled, the conditions for the definition of a constellation mission within the MMOS are met, as are the conditions for the composition of a Constellation Planning Tool (CPT) within the MPS. How this can be done shall be elaborated conclusively in the following. This will further show how the requirements derived from the reference architecture (sec. 3.2) are met by the concept.

6.5.1 Layout

A CPT consists of multiple Mission Planning Tools. It is therefore more of an assembly rather than a tool. Following the terminology introduced in section 5.2.1.1 it should actually be called a constellation planning *unit*. The term CPT was chosen though to avoid confusion with the general understanding of the acronym *CPU*.

An example CPT configuration is shown in figure 6.27.

Every satellite within the constellation is represented by its own MPT. As usual, the satellite schedule is uplinked via its dedicated MCT instance. Therefore, each satellite in the constellation can be operated individually as discussed, either manually by a human or automatically by a satellite Agent. The individual satellite MPTs keep track of the schedule and the resource household, regardless by whom Activities have been scheduled.

If one recalls one of the major objectives formulated in section 3.1, an envisaged quality of constellation operations was the treatment of the *entire* distributed system as a whole, rather than the capability of operating n satellites individually. As a consequence, a CPT must feature an entity that merges the separate satellite Activities into one single constellation schedule. That entity is the *constellation MPT* (fig. 6.27). The constellation MPT does not manage individual satellite activities, therefore no commands are extracted from that schedule. Instead, the constellation schedule contains those parent activities, who's children are executed by the different satellites. The child activities are managed individually by the junior satellite MPTs. Commands from these schedule are extracted as explained in section 6.4.

As mentioned many times, the Activity concept allows for the decomposition of a (parent) task into various subtasks (children). The CPT takes advantage of that. Whenever an entity requests a constellation Activity, a global parent task is scheduled within the constellation MPT, while subtasks must be scheduled within each

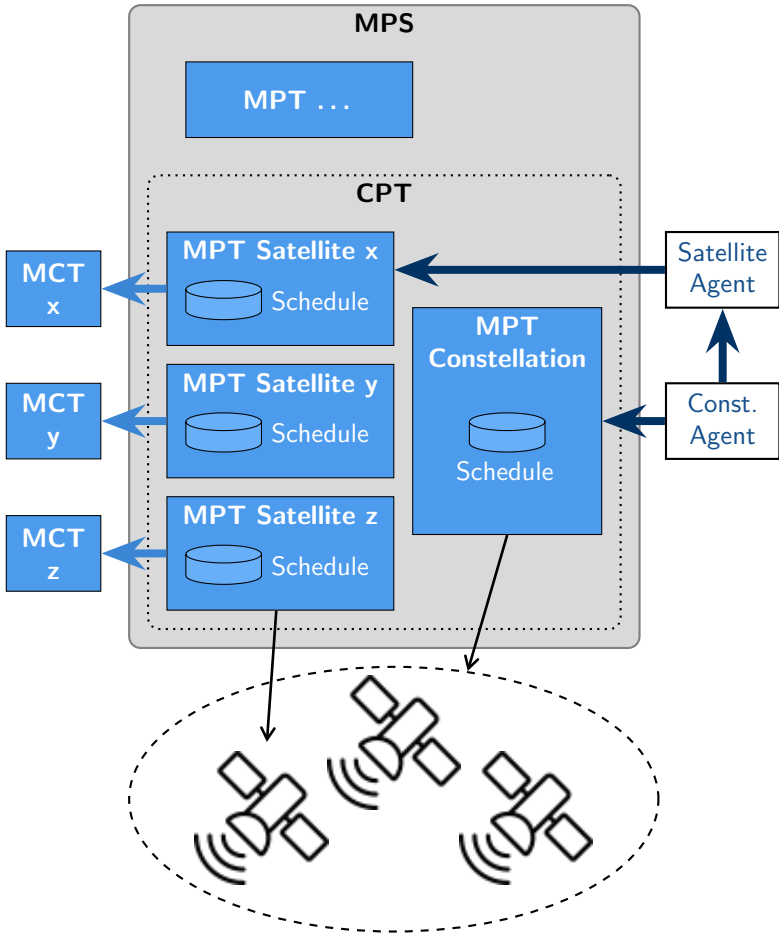


Figure 6.27: Mission Planning Layout for the Realization of a Constellation Planning Tool

satellite MPT. Purpose of the constellation MPT is the verification of the parent task, through monitoring of those child activities executed by each satellites. This approach enables a human operator to follow the state of a constellation Activity without the necessity of examining each involved satellite individually. In case of a failed execution though, the hierarchical parent-child nature of Activities enables an operator to trace the cause of the problem back to the satellite where the error occurred.

Within the automatic control process, MPTs are the implementation of the response action mechanism (fig. 5.20). They do not provide means of system-level decision making. Any decision making must be implemented by the Agents.

Purpose of the Agents is the creation of the constellation Activities and the distribution of child tasks among the satellite schedules. Making a decision in terms of which satellite does what and when is generally a non-trivial task though. The more complex the mission becomes, and the more satellites compete for limited resources (e.g. time slots for ground station passes), the more likely those tasks become the matter of optimization problems [92].

Areas in which Agents can be used to implement the decision making process are manifold. Classical applications in constellation operations are for example:

- GNC and maneuver planning
- collision avoidance
- pass planning and link scheduling
- payload operations and data downlink scheduling

6.5.1.1 Implementation

Figure 6.27 displays the CPT from a data flow perspective. It has been emphasized that each satellite schedule, as well as the constellation schedule are managed by a separate MPT. That raises the

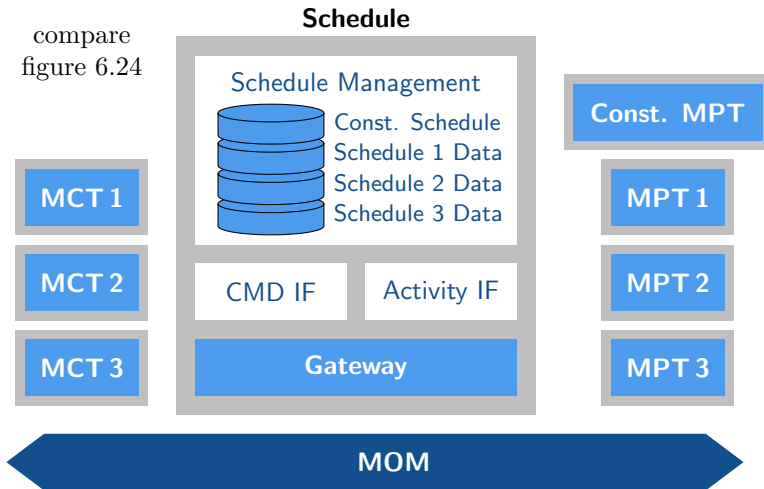


Figure 6.28: Implementation of the Constellation Planning

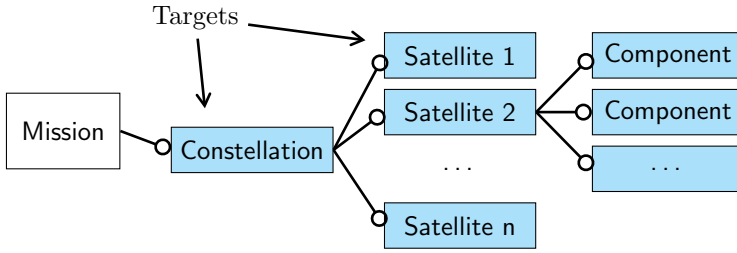
question how such a setup is implemented and how a data flow between schedules and the different MPTs can be established.

The integration of schedule and MPT for a single system has been introduced in section 6.4. Implementing a CPT does not work any way different. It means hosting multiple MPTs, and connecting them to the MOM as usual. Via the MOM, the MPTs can request schedule data as described above. Yet, instead of one single schedule, the schedule management now maintains databases for each operated satellite, plus one constellation schedule containing the parent activities (fig. 6.28). This way, a constellation MPT can obtain information about each satellite schedule by making a request at one central component.

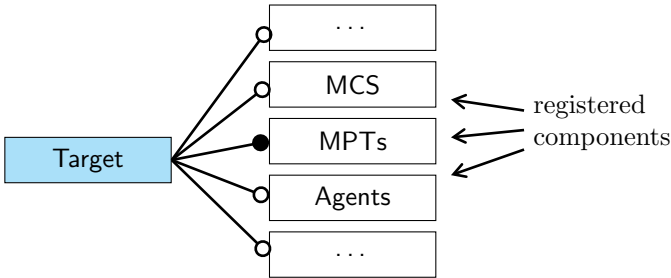
The release of commands via the individual MCTs works in the same way as described in section 6.4.

6.5.1.2 Orchestration

A CPT layout as depicted in figure 6.27 needs to be orchestrated during launch of the MMOS. Again, orchestrating a distributed sys-



(a) Decomposing a Mission Into Different Targets by the SIB



(b) Registering MMOS Components for Each Target

Figure 6.29: The definition of multiple targets within one *mission* and the registration of MMOS components follows the concept of the SIB, which has been introduced in section 5.3.3.1. See also figure 5.34.

tem works not fundamentally different from orchestrating a *normal* layout.

All information about which components must be instantiated and registered at the middleware come from the SIB (sec. 5.3.3.1). Its format allows decomposing a mission into a tree of so called *targets* (fig. 6.29a). The difference compared to a single-satellite mission is that a constellation demands multiple of these targets being specified. All required MMOS components can be specified for each target individually (fig. 6.29b).

Basically, every hierarchical system can be decomposed in this manner, and be represented by the MPS. It does not necessarily have to be a constellation. Already a single satellite consists of several

subsystems and components, which could be represented, scheduled and commanded independently from different MPTs.

6.5.2 Reflection of Architecture Requirements

In section 3.2 a reference constellation architecture has been discussed against which the MMOS has been designed. The following recapitulation briefly shows how the characteristics of that reference architecture are reflected by the MMOS, the CPT, and the proposed Activity concept.

6.5.2.1 Constellation Size

The number of satellites that can be registered at the MMOS is limited by the address range, and the number of components and entities also registered at the system. A 16-bit address range was considered appropriate to register multiple thousand MPTs at the middleware, which is more of a theoretical scenario though. Thus, the address range is probably not the limiting factor here.

Instead, the scaling capabilities of the MMOS are limited by the computing performance of the underlying hardware, the available server size, and the performance of the middleware.

6.5.2.2 Operating Principle

In section 3.2.3 it has been stated, that a constellation can achieve a mission goal in two different ways. In a *symbiotic* system, each satellite fulfills a dedicated subtask of a top level goal. The overall goal can only be achieved if all satellites perform correctly. *Collaborative* systems instead are systems in which each satellite performs individually, and it does not matter by which satellite a task is performed [117]. Large fleets of earth imaging satellites are a good example for the latter.

Both operating principles are supported by the mission planning concept.

The distribution of tasks amongst the satellites of a collaborative system is rather simple. Under certain circumstances, it does not even require a constellation MPT, because the individual Activities are executed independently or autonomously by the satellite [103].

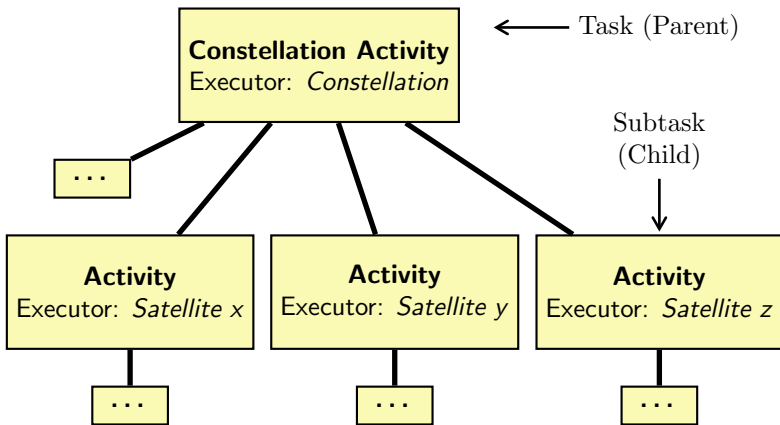
Symbiotic systems however require a correlation of satellite Activities. This is why representing such a system is not really a matter of the CPT orchestration, but the distribution and the correct nesting of Activities. Two examples are shown in figure 6.30.

6.5.2.3 Homogeneity, Distribution and Service Availability

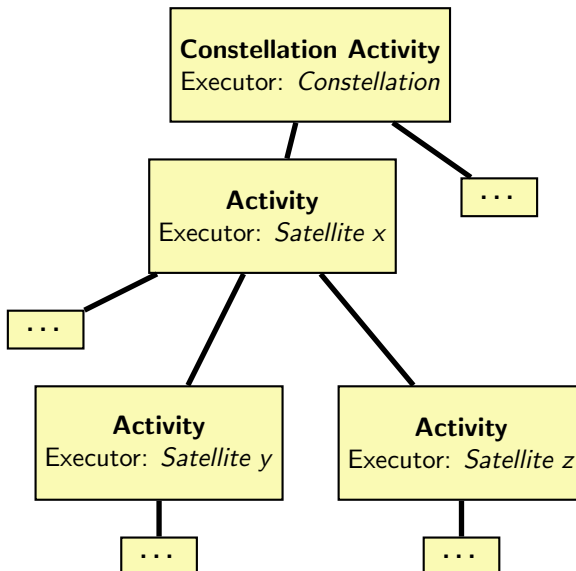
Because, every satellite is registered as a standalone mission in the SIB, and since each satellite can be addressed and planned individually, constellation planning does not make any restrictions in terms of the constellation homogeneity. Consequently, the MMOS supports constellations of identical satellites, as well as inhomogeneous systems.

The proposed scheduling concept further makes no restrictions in terms of the spacial distribution of the constellation (or the formation). Naturally, operations of satellites in close proximity goes along with a series of challenges. But those are commonly solved by on-board autonomy or they need to be handled within the decision making process. For example, the scheduling of passes is more problematic, if the satellites competing for link time always appear at the same time. The problem of course depends on the specific orbit and shall not further discussed here.

Furthermore, no restriction is made in terms of the availability of the constellation. The proposed concept is indeed designed for the scheduling of Activities that are executed when there is no connection to the satellite, but it can of course be applied to contact-only operations as well.



(a) Example I - Simple distribution of subtasks



(b) Example II - Certain satellite activities require the successful execution of further child activities by other satellites. (e.g. Relay satellite x can only forward image data, if satellites y and z take pictures.)

Figure 6.30: Correlating nested Activities in a Symbiotic System

Systems that rely on inter-satellite links are also covered, as the link Activity does not dictate how the link between two communicating systems is established. The concept only demands that each system actively commanded for the establishment of a link is represented by an MPT.

System Test and Verification Environment

One of the design requirements of the MMOS was reusability (sec. 5.1.2.1). Consequently, the system has been designed in a way it can be configured for a new mission easily. However, the compatibility of the configured Operating System (OS) with the operated system still needs to be verified in each individual case, which is why appropriate test means are necessary. A suitable test facility for the system had been designed and integrated already prior to the development of the MMOS itself.

The development of a verification architecture was also driven by the multi-mission character of the to be tested OS. This is why reusability was one of the quality criteria also for the simulation infrastructure. New simulator models of a system added to MMOS should not have to be developed from scratch.

This chapter introduces the architecture and the most important elements of the MMOS test facility. Core of that facility is the satellite system simulator, which processes the MMOS outputs and generates its inputs. Fundamentals of system simulation and its value for software development were discussed in section 4.5.3

already. The following sections cover some specifics of the implementation and how they support the idea of multi-mission use.

7.1 System Simulation

Purpose of system simulation as described here is the modelling of the state of an operated system (usually a satellite) in time. This is realized through numerical simulation models, implementing the temporal behavior of a component, an entire system, or a physical phenomenon [47]. Several aspects of the system simulator realization are discussed in the following.

7.1.1 Model Libraries

The general problem of a verification approach with a simulator in the loop is the time consuming development of a sound system simulation. Why is that?

Developing a simple simulation model is indeed very easy, especially if one is supported by a modelling framework like SimMF. If however a certain model fidelity is required, the implementation efforts can increase superlinearly so that in the end the costs can outweigh the benefit of the simulator.

Furthermore, a sound system simulation is only possible, if the simulator has reached a certain degree of implementation. This is because the simulation of a certain phenomenon does not only depend on the availability of a number of models, it also depends on the interconnection of these models. If that interconnection is not implemented correctly, or just one model in the chain is missing, the entire simulation can fall apart. The fact that the simulation of a phenomenon, a component, or a system normally depends on a number of interconnected models, makes system simulators themselves extremely complex and difficult to test (sec. 4.5.1.3).

This is why, efforts of realizing a system simulator should be initiated already during the very early stages of a satellite mission project. On the other hand, a sound simulation is only possible if enough knowledge about the to be simulated system is available, knowledge that usually has not been gained during early project stages. That discrepancy and the fact that a simulator is an additional, costly development in parallel to the actual satellite project, can result in a low acceptance for a software based system simulator, especially in smaller projects.

These problems can be solved by adopting some basic design principles. First of all, models should be kept as simple as possible. Not everything that can be simulated also should be simulated. Secondly, not every model needs to be designed from scratch. For almost every component type or physical phenomenon, modelling approaches exist which are sufficiently accurate for a system modelling.

In order to support future missions with the development of a system simulator, efforts were made to reduce model implementation cycles. One result of these efforts has been a library of generic models that later mission specific models can build on.

The model of a satellite component is indeed highly mission specific. Nevertheless, every component, for example a reaction wheel, features certain qualities that can be simulated by means of generic physical models, in this case: a rotating mass body. Implementing a mission specific device is then just a matter of wrapping device specific functionality around the basic physical model.

As explained in section 4.5.3.4, modelling describes the process of mapping certain qualities of a real object into an artificial representation of the object. Depending on the kind of object and the use case of the model, the number of qualities that need to be simulated varies. Within satellite system simulation, only a limited number of physical qualities of a component are usually of interest. These are:

- geometric qualities (position, attitude, ...)
- kinetic qualities (mass, inertia, ...)
- electric qualities (internal resistance, capacity, ...)
- thermal qualities (heat capacity, emissivity, absorptivity, ...)

Intelligent devices with built-in controllers further require pure functional models, which simulate the *observable* behavior of the integrated circuits, for example the reaction of a command, or a boot process. As these models simulate very specific qualities of the device, they usually cannot be generalized.

In the following section the scope of the modelling of the relevant physical phenomena shall be introduced.

7.1.1.1 Environment and Dynamics Modelling

The Environment and Dynamics Models (short: EnvDyn) refers to a group of models in charge of calculating position and attitude of the simulated object in space. Core of the EnvDyn is a structure model, representing the mass and inertia properties of that object (e.g., a satellite).

In orbit, the satellite structure is exposed to several forces and torques. Some of these are generated by the satellite actuators, and some are due to the space environment. Purpose of the EnvDyn is further the simulation of those forces and disturbing torques, induced by the environment. These are generated e.g. by:

- the earth gravity and the gravity gradient,
- the residual atmosphere,
- the gravity of the moon (only on higher orbits).

The EnvDyn determines satellite position, attitude, and velocity vectors of the satellite through the integration of all attacking forces and torques by means of the Runge-Kutta Method [32] (fig. 7.1).



Figure 7.1: Inputs and Outputs of the Environment and Dynamics Simulation

Position and attitude are important inputs for many other models, whose functionality depend on that information (sensor models, solar cell models, etc.). But also the EnvDyn itself needs these simulation results, since the disturbing environmental forces are again functions of the current position and attitude of the satellite [108, 7].

All computations related to position and attitude are performed with respect to an inertial reference frame. For those models which require the simulation results with respect to a different coordinate frame, the EnvDyn implements the conversion. Relevant coordinate frames are the following:

- earth-centered inertial (ECI) (*used for computation*)
- earth-centered earth-fixed (ECEF)
- latitude, longitude and altitude (LLA)
- body-centered inertial (BCI)
- body-centered body-fixed (BCBF)

7.1.1.2 Mechanical Modelling

Purpose of the mechanical modelling is the representation of the geometrical and mechanical properties of a component. Modelling these is necessary when they have a significant effect on the rest of the simulation, which for instance is the case for moving parts, or

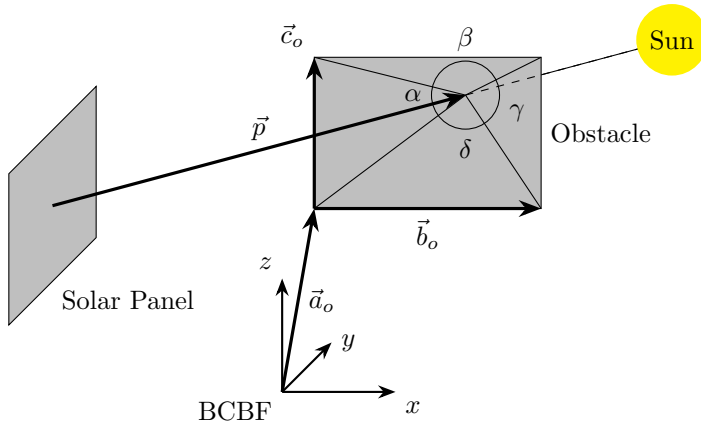


Figure 7.2: Simulation of an Obstacle Through the Simulation of a Geometrical Plane - The solar panel is shaded, when the sun vector intersects the obstacle plane within the gray area.

parts with time dependant mechanical properties (e.g. fuel). Components rigidly connected to the satellite structure do not demand a dedicated mechanical simulation. They are normally considered by the mechanical model of the main structure already.

Moving parts must be simulated, because a deployment has an impact on the center of gravity and the inertia tensor of the satellite. The mass and inertia properties of moving part models are an input for the EnvDyn, because for the simulation of the satellite motion all mass and inertia properties must be condensed into a single tensor (fig. 7.1).

An example for the use case of a geometrical model is shown in figure 7.2. If an object (e.g., another solar panel) is located in the incidence range of a solar panel, geometrical models can be used to simulate shading effects.

7.1.1.3 Thermal Modelling

Purpose of thermal models is the determination of the temperature distribution within the satellite structure. The only transfer mech-

anism that applies in space for the heat exchange between satellite and space environment is radiation. In earth orbit, radiation that encounters the satellite is the visible radiation from sun, the earth albedo radiation and the earth infrared radiation ([134], p. 689, fig. 22-17). The satellite body itself emits black body radiation towards the space environment with a background temperature of almost $0 K$.

Satellite structures are simulated thermally through a network of thermal nodes, where each node represents the thermal properties of a component with a heat capacity c and a temperature T . Within that network, nodes exchange heat P through radiation or conduction. Active nodes, for example electrical devices, also dissipate heat. This means that for each node the following simplified equation must be solved [122]:

$$P_{\text{sol.}} + P_{\text{alb.}} + P_{\text{e,IR}} + P_{\text{diss.}} - \sum P_{\text{i,rad.}} - \sum P_{\text{i,cond.}} = m c \frac{dT}{dt} \quad (7.1)$$

7.1.1.4 Electrical Modelling

Almost every component in a satellite processes electrical energy. Purpose of the electrical modelling is the simulation of the electrical energy household of the system. For the simulation of such, each simulated device is either simulated as an energy *consumer*, or an energy *provider*. A consumer model, simulates the device current consumption as a function of the supply voltage: $i = f(u, t)$, where a provider model simulates the supply voltage as function of the consumed current: $u = f(i, t)$.

The electrical models are connected via interface models simulating the power lines. Those line models are used for the exchange of information on the consumed current, or the available supply voltage.

Basis for each electrical model is an electrical replacement diagram, like the mesh in figure 7.3. Within the model such a mesh is

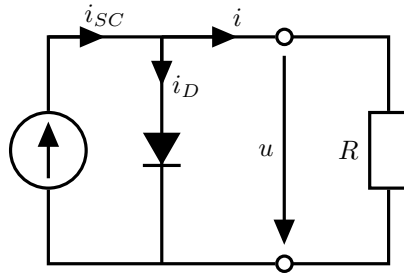


Figure 7.3: Electrical Mesh of a Solar Cell

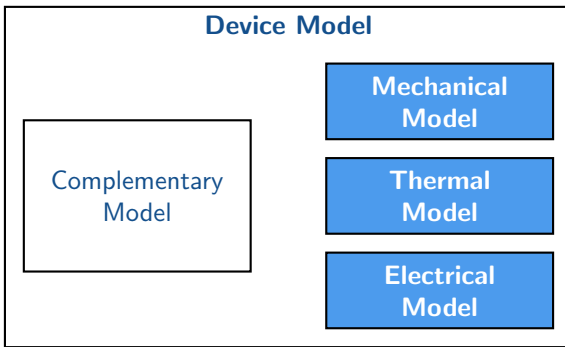


Figure 7.4: Device Model - A device can be modelled by means of generic physical models plus one or more complementary models.

expressed by the appropriate differential equation, which is solved numerically during each model step. If the elements in the mesh (resistors, diodes, etc.) have a strong temperature dependency, or when the component is exposed to large temperature changes, simulating electrical devices also requires a thermal modelling too.

The model library features generic electrical models of batteries and solar cells for reuse in different simulation projects.

7.1.1.5 Device Model

Through the combination of different physical models, a representation of a real device can be compiled. In practice, though, a component always features certain characteristics that go beyond those

generic physical properties. This is why almost every model must be complemented by one or more device specific models (fig. 7.4). Complementary models can be used for:

- simulation of complementary device features
- simulation of controllers, logical components or state machines
- simulation of interfaces
- simulation of buses
- model initialization
- creation and configuration of assemblies

While generic physical models can be provided by the library, device models, respectively the complementary models, need to be implemented for each simulation project individually.

7.1.2 OBC Emulation

In section 4.5 some of the general problems of software testing and verification were addressed. One of these problems is that testing a component in a real working environment is usually not possible, which can be due to several reasons:

- a test in a real environment would be too expensive
- a test in a real environment would destroy the specimen
- a test in a real environment is technically impossible
- elements of the real working environment are not available for the test

A common solution to this problem is to mimic the environment as good as possible by means of a simulator (fig. 4.14). Scope, fidelity, and design of the simulator are driven by the complexity of

the test and by the qualities of the tested item. Depending on what needs to be tested, the approaches can be manifold. For instance, an astronaut's training requires hours in a centrifuge simulating a multiple of the earth gravity in order to test the astronaut's capabilities to withstand the loads during launch and reentry. The capabilities of technical components to withstand a use in space is tested by putting them item into a thermal vacuum chamber simulating the space environment and the radiation conditions ([47], p. 19).

Purpose of the simulator described here is to enable End-to-End (E2E) tests of the MMOS, with and without human operators in the loop. These tests aim for the verification of the entire workflow, by ensuring "that [the] completely integrated system works together from end to end" ([93], p. 183).

Such an E2E test requires a full simulation of the satellite system from a data handling perspective. What does that mean? The counter part of the MMOS in space is the satellite's On-board Computer (OBC), respectively the on-board software. Both systems communicate with each other on the basis of TM and TC. An E2E therefore requires that the OBC including its software is part of the simulation, so that all TCs can be processed and the expected TM is generated and returned. Apart from that, the simulated system shall behave exactly like the real one, which is achieved by the other component models.

Simulating computer architectures is commonly referred to as *emulation*. Respective OBC models are fundamentally different though, simply because the model must be capable of executing an on-board software binary. A modelling approach as the one described above is not capable of doing that. In the scope of this work several opportunities have been evaluated to integrate an emulator into the system simulation [86].

Linux Binary

The OBC of a satellite is an embedded system. Systems like these usually rely on lightweight real time operating systems, which are designed for this kind of application. The on-board software framework used for this test is built on RTEMS [109, 9]. The idea was to take the framework and modify it in a way that it can be executed in a Linux system instead [139]. So, on-board software and simulator would have been hosted and executed in parallel by the same machine without the need of an extra emulation environment. First developments in that direction turned out to be promising.

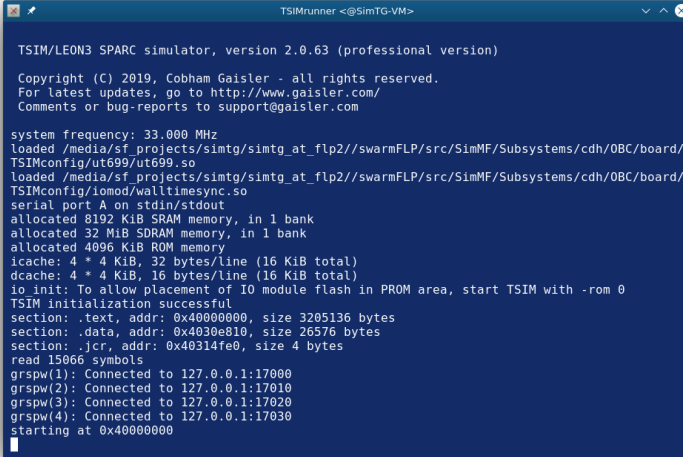
The disadvantage of this approach is the low flexibility though. Instead of one, two on-board software versions need to be maintained, and it must be ensured that the development of the Linux variant takes pace with the development of the actual flight version. Furthermore, the solution cannot guarantee that the modified on-board software behaves exactly like the original in any given situation.

Open Hardware Emulator

An alternative to the approach described above is the use of a hardware emulator, which simulates the used CPU architecture, and executes the on-board software in a virtual environment. Such environments are referred to as *virtual machines*. A series of open and proprietary solutions are available for the purpose. The selection of one depends on the to be simulated computer architecture, the application, and in the end the required fidelity/accuracy of the emulation.

A popular, open solution for the emulation of embedded systems is QEMU, which supports a variety of platforms like ARM, i386/x86-64, or Sparc [10].

The advantage of free hardware emulators like QEMU, libvirt, Docker, etc. is that they are used by large communities. These communities can offer support, report bugs, and thus contribute to the emulator development.



```
TSM/LEON3 SPARC simulator, version 2.0.63 (professional version)

Copyright (C) 2019, Cobham Gaisler - all rights reserved.
For latest updates, go to http://www.gaisler.com/
Comments or bug-reports to support@gaisler.com

system frequency: 33.000 MHz
loaded /media/sf_projects/simtg/simtg_at_flp2/swarmFLP/src/SimMF/Subsystems/cdh/OBC/board/
TSMconfig/ut699/ut699.so
loaded /media/sf_projects/simtg/simtg_at_flp2/swarmFLP/src/SimMF/Subsystems/cdh/OBC/board/
TSMconfig/omod/walltimesync.so
serial port A on stdin/stdout
allocated 8192 KiB SRAM memory, in 1 bank
allocated 32 MiB SDRAM memory, in 1 bank
allocated 4096 KiB ROM memory
icache: 4 * 4 KiB, 32 bytes/line (16 KiB total)
dcache: 4 * 4 KiB, 16 bytes/line (16 KiB total)
io init: To allow placement of IO module flash in PROM area, start TSIM with -rom 0
TSIM initialization successful
section: .text, addr: 0x40000000, size 3205136 bytes
section: .data, addr: 0x4030e810, size 26576 bytes
section: .jcr, addr: 0x40314fe0, size 4 bytes
read 15066 symbols
grspw(1): Connected to 127.0.0.1:17000
grspw(2): Connected to 127.0.0.1:17010
grspw(3): Connected to 127.0.0.1:17020
grspw(4): Connected to 127.0.0.1:17030
starting at 0x40000000
```

Figure 7.5: Proprietary SPARC LEON3 Emulation - Start-Up Sequence

Proprietary Hardware Emulator

If the simulation project cannot rely on public community support only, or if the used architecture cannot be emulated sufficiently with a freely disposable solution, a proprietary hardware emulator is needed. Proprietary emulators are normally provided by the manufacturer of the processor boards, and allow for the accurate emulation of the used CPU. A sound emulation, as well as customer support, normally comes at the price of a medium sized car though.

In this work, several licensed derivatives of the Scalable Processor Architecture (SPARC) are simulated, which are used by *Flying Laptop*. Four licenses for such an emulator were provided by research partner Airbus and the manufacturer of the processor board. A start-up sequence of that emulator is shown in figure 7.5.

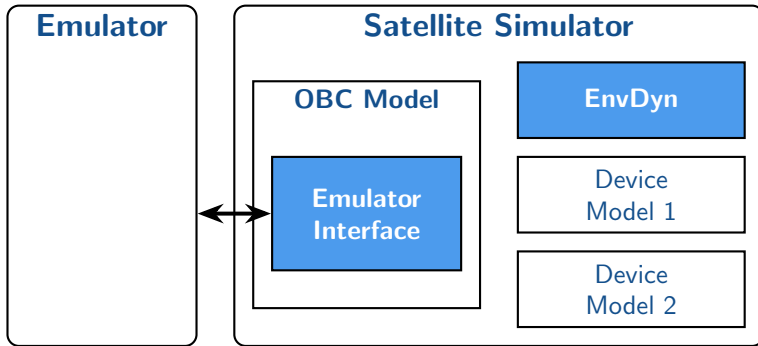


Figure 7.6: Setup of the OBC Model within the Simulator

7.1.2.1 OBC Model Integration

With the emulator in place, an OBC model can be integrated into the simulation. At first, the OBC model is integrated into the simulator like an ordinary model. As such it can feature a thermal model, a rudimentary electrical model for the simulation of power consumption, or any sort of complementary model.

Besides that, the OBC model features an interface that controls the emulator. The emulator, as described above, runs within a dedicated process outside the actual simulation (fig. 7.6). Purpose of the emulator interface is to initialize, start & stop the emulation, and to keep it in synch with the simulator. How the latter is achieved is shown in figure 7.7.

Described here is a synchronous simulation approach, which means that each model in the simulator, respectively its step function (sec. 4.5.3.1), is called periodically by the scheduler in the SimTG kernel. The period dt of these calls is an individual constant for each model.

With each call of the OBC step function, the model advises the emulator to run for another time step dt . The problem is that the Zulu duration \tilde{dt} of an incremental emulator run is neither equal to dt , nor constant, which is due to two reasons:

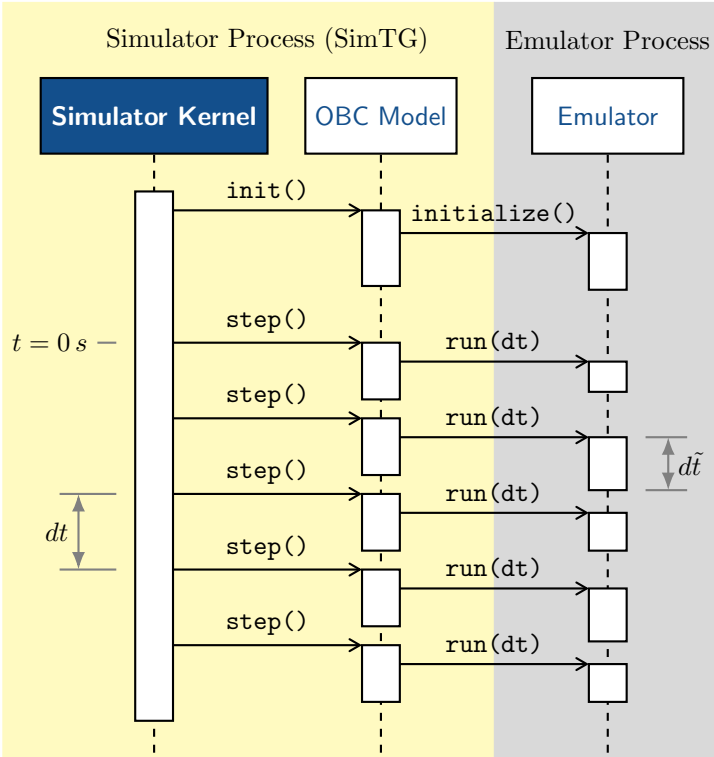
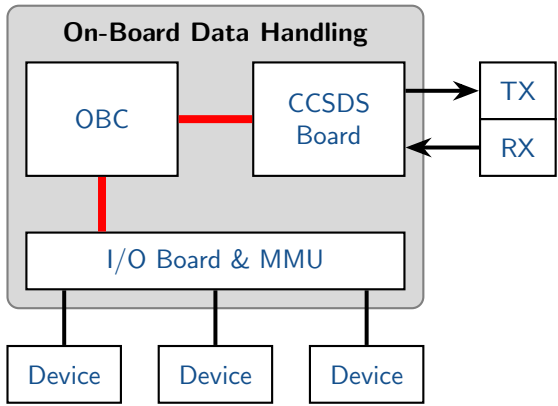


Figure 7.7: Emulator Synchronization - The sequence diagram shows how the emulation is synchronized with the simulator by means of the OBC Model. `init()` and `step()` are function calls that the simulator makes at the instantiated OBC model. The calls the OBC model makes at the emulator are no function calls in the classical sense. Instead, `initialize` and `run(dt)` are commands sent to the emulator via socket. So, the OBC model does not expect a return from the emulator.

1. With each incremental run, a series of instructions are executed by the emulator. If an instruction is started within the time frame dt , that instruction will be executed although its execution overshoots the scheduled duration. If the emulation is not actively paused after dt , simulation and emulation drift apart in time.
2. By default, the emulation runs as fast as possible, which means that the simulation of a time period can happen significantly faster than real time ($d\tilde{t} \ll dt$). The speed factor by which the emulation is faster than real time can be set. If it is set to 1, the emulator doesn't run faster than Zulu time.

So, the expected behavior would be that with a speed factor set to 1, emulator Zulu time and Simulation Runtime (SRT) would be synchronized, but unfortunately this is not the case. The speed factor only limits the execution speed to the top. It does not prevent the emulation from being slower than real time, which can happen easily. To circumvent a drift between simulator and emulator, caused by the emulator being too slow, a speed factor slightly faster than real time has been selected. The selection of the right parameter is quite a trade-off. The crux is on the one hand to minimize the gaps between termination and restart of the emulator execution, and on the other hand to avoid overshoots.

Why is keeping simulator and OBC emulation in synch so important? The modelled OBC uses SpaceWire for the communication with its peripherals. Unlike the OBC CPU, those peripherals are modelled in the simulator. A sending SpaceWire node, expects a response from its target within a few microseconds [39]. If the emulation runs at an accelerated speed, compared to the rest of the simulation, the emulated OBC SpaceWire core would always detect a timeout when trying to establish device communication.



Legend






-
-  Subsystem
 -  SpaceWire + RMAP
 -  Model
 -  Frame
 -  Device IF

Figure 7.8: Simplified Model Setup of a On-Board Data Handling System

7.1.2.2 Data Handling Simulation

Figure 7.8 illustrates the simplified setup of the on-board data handling subsystem, as modelled within the simulator. Core of that subsystem is the satellite OBC, which is modelled and integrated into the simulator as described above. The OBC model receives TCs from a CCSDS board model, which decodes the TC frames from a receiver model. In turn, the OBC produces TM, and sends it back to the CCSDS board. The CCSDS board then composes TM frames, which are returned via the transmitter model eventually.

Via an I/O board model, various device models are connected to the OBC. The I/O board buffers the in- and outgoing device messages and features the Mass Memory Unit (MMU) of the data handling system.

Within this data handling system, a SpaceWire Network facilitates the communication between the three nodes: OBC, I/O board, and CCSDS board. In- and outgoing messages (device messages or ground TM/TC) are buffered within simulated memory blocks in the boards. By means of the Remote Memory Access Protocol (RMAP) [40], the OBC is capable of reading and writing these blocks, and thus to communicate with ground, or with the devices on board the satellite.

As mentioned in section 7.1.2.1, the problem of routing traffic between simulated devices and a SpaceWire core (real or emulated) are the harsh time constraints of the protocol. A synchronous model scheduling as described in section 4.5.3.5 is simply too slow for the simulation of a SpaceWire bus. This is just one example. The same issue can occur with other bus systems as well, which is why the following solution can be considered universal and not just for a SpaceWire application.

Assumed is the following situation (fig. 7.9): Via Model B, a command shall be routed from Model A to Model C, and A expects

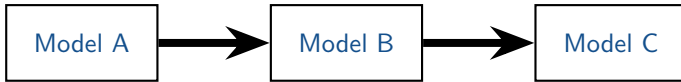


Figure 7.9: Data Transmission between Device Models

a response from C within microseconds. How can this be achieved, if the model scheduling frequencies are too slow for a response in time?

The simplest solution could be increasing the model scheduling frequencies, but this would be rather inefficient. Furthermore, the scheduling capabilities of SimTG are limited. Model frequencies in the order of 50 Hz and higher caused larger simulations to crash regularly.

A better approach is to handle the data packets asynchronously. Therefore, each model within the data handling system was extended by an asynchronous layer. Purpose of this layer is to wait for incoming data from a source and forward the data to a sink as fast as possible. Sources and sinks can be:

- model interfaces
- device memory (as part of the model)
- sockets (e.g. needed to route packets from and into the emulator)

Figure 7.10 indicates how such an asynchronous layer is integrated into a synchronous model. A device model, equipped with such a layer features a handler, which is nothing more than a thread, continuously waiting for data at the source. The handler is controlled from the model step function as part of the functional simulation. For instance, if the simulated device is off (zero voltage on the power input), the handler is paused by the model and all incoming data will be discarded. If the device is active, the handler is resumed again and incoming data will be forwarded to the sink.

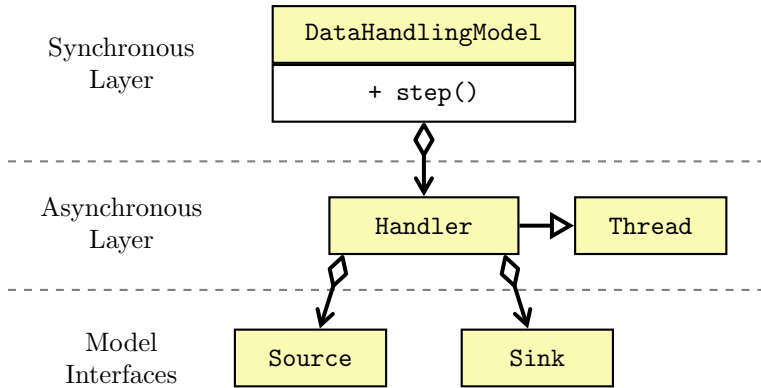


Figure 7.10: Class Diagram of the Asynchronous Layer in a Data Handling Model

Since the data forwarding is executed by an additional thread, the fidelity of the data handling is no longer limited to the scheduling frequency of the model. Furthermore, data forwarding and the functional behavior of the device model are decoupled. What does that mean? The only purpose of the handler is transmitting messages from source to sink. As long as the sink is not within the model, the data forwarding won't affect the functional behavior of the device, as modelled within the step function.

Unfortunately, the solution comes at the expense of two further problems. The first one is that a former single-threaded simulator becomes multi-threaded. Multi-threading a software always requires a lot of care by the programmer, especially when those threads access the same memory regions by design. If that is the case, the software must be prevented from segmentation faults. A common means of doing that in the C++ programming language are mutexes.

The second problem is a direct cause of solving the first one, which is that the models implementing such an asynchronous layer are no longer SMP2 compliant. The loss of SMP2 compliance was accepted though, because the affected models were highly mission specific anyway.

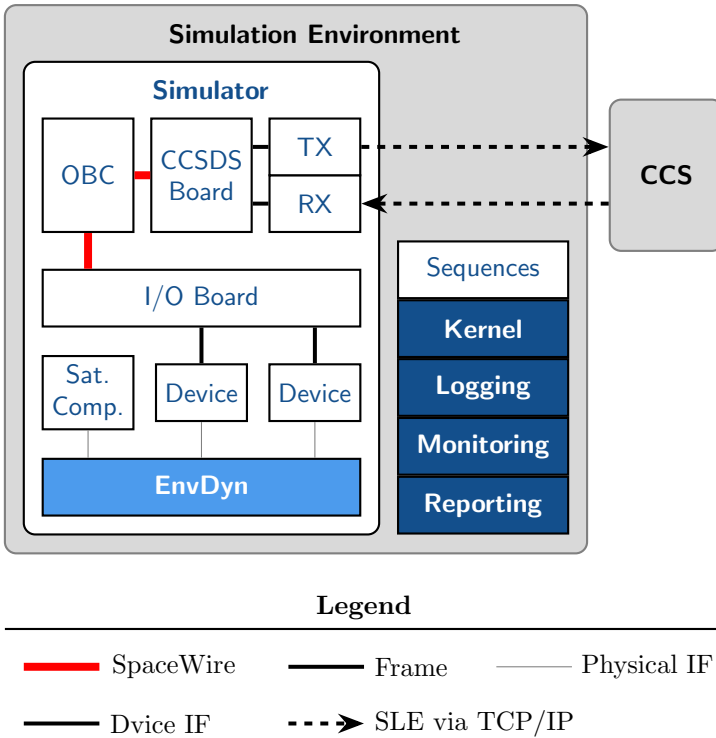


Figure 7.11: Setup for Model and Simulator Checkout - Simplified schematic of the simulated satellite bus, connected to a Central Checkout System (CCS). For convenience, redundant component models are not displayed.

7.2 Simulation Infrastructure

A simplified schematic of the simulator as developed and used within this work is shown in figure 7.11. Core of it is the simulation of a data handling system as described in section 7.1.2.2.

Via two antenna models (TX and RX) the simulator can exchange TM/TC with a commanding entity. The telecommands are executed by the on-board software on the emulated OBC. Simulating the various satellite functionalities (e.g. attitude control) re-

quires the simulation of all devices involved. If the satellite receives a command to maintain a certain attitude, the OBC engages an on-board control process. That requires actuators being commanded to produce the appropriate torque. Actuator models, which simulate the device behavior and calculate the generated torque, are the recipients of such on-board commands. The produced torques are integrated by the EnvDyn as described in section 7.1.1.1. The attitude calculated by the EnvDyn is returned to the OBC via the appropriate sensor models.

Thus, a complete control process is implemented within the simulation and an observer, communicating with the simulated satellite via TM/TC should not notice a difference to the real system. The same concept of course applies to other control processes and functionalities on board the satellite.

7.2.1 Simulator Checkout System

A system simulator is a complex software. So, before it can be used as part of a verification environment (fig. 4.14) it must be tested itself.

Verification of a simulator follows the same principles as introduced in section 4.5. According to the traditional V-model approach, this covers method, class and unit tests by the developer, followed by model integration tests, and simulator system tests eventually.

The selected simulation IDE supports the developer in the model testing, by providing functionality for test implementation and automation. For instance, method and JUnit tests can be triggered as part of the simulator compilation process. More complex tests require the execution of sophisticated simulation sequences stimulating the model input interfaces and verifying the model output.

In a next step, clusters of interacted models are tested. Again, this is achieved by means of simulation sequences (sec. 4.5.3.3). The more models are part of these tests, the more complex they become

and the longer it takes to create and execute the sequences. Due to the complexity of these tests and the nature of object-oriented programming, those tests cannot guarantee a complete verification of the simulation though [121].

At some point the checkout of the simulator involves the OBC model. The OBC is usually checked out last, because it requires the presence of all the other models to function properly. As mentioned above, the on-board software expects commands from ground so that certain of its functionalities are getting triggered. So, a simulator test involving the OBC actually requires interaction of two different software products: the simulator, and a ground entity sending TCs and processing TM.

If the simulator is the tested item, verification is only possible if the commanding entity has been formally tested. Otherwise, mutual consistencies might impede the detection of bugs at one or the other side. Which is why a commercial CCS was selected to support the testing of the simulator. The setup is shown in figure 7.11.

In a later use case of the simulator that CCS is about to be replaced by the MMOS, which is then the tested item.

7.2.2 System Simulation Network

The verification of a multi-mission OS of course requires the simulation of a number of satellites. Therefore, an infrastructure was set up supporting the instantiation of several of such simulators in parallel (fig. 7.12).

Core of that simulation environment is a server architecture as specified in table 7.1. Installed on that server is a libvirt virtualization environment¹ by means of which a number of CentOS Linux VMs were created, each running an instance of SimTG. By checking out simulation projects from a central Git repository into the

¹Server installation and the setup of the libvirt environment were not part of this work.

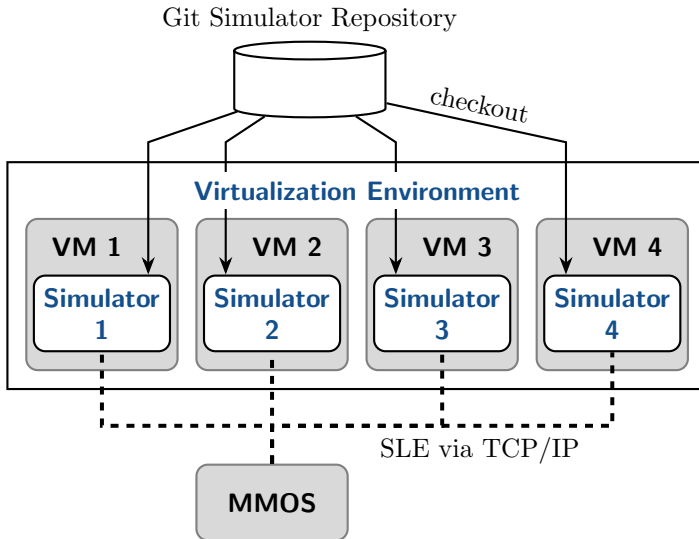


Figure 7.12: Network of Multiple Simulated Satellites - Each simulator is executed by a dedicated Virtual Machine (VM), hosted in a libvirt environment.

Table 7.1: Feature List of the Virtualization Server Hosting the Simulation VMs

Feature	Description
CPU	2x 16 Cores, 2.4 GHz, 64 MB Cache
Memory	2x 32 GB DDR4
Network	2x GBit LAN
Mass Memory	240 GB (SSD) + 1 TB (HDD)

VMs, various simulators can be executed, each simulating exactly one spacecraft. The complete setup allows for the simulation of multiple satellites in parallel, as well as for the simulation of constellation scenarios.

Via Gigabit LAN, the Virtual Ground Stations (VGS) within the MMOS MCS (fig. 5.12) can connect to the RX/TX models (fig. 7.11). Purpose of the RX/TX models is then the simulation of the entire TM/TC chain between the MMOS and the satellite's transceiver. Both parties of course must then implement the same protocol. In the example in figure 7.12 this is ESA Space Linke Extension (SLE) via TCP/IP, but it can be any other protocol as used for the mission's TM/TC routing on ground.

7.3 Simulated Mission

Goal of system testing is to verify the constellation operation capability of the MMOS, as well as the verification that it is really capable of handling multiple missions simultaneously. Simulator design and test scenarios must therefore aim for the testing of those hard functional requirements discussed in section 5.2.2, as well as for the testing of weak quality requirements specified in section 5.1.

With all models and a simulation infrastructure in place, an appropriate simulation environment for the MMOS system test could be set up. Such an environment shall only feature formerly verified satellite system simulators with verified on-board software, as it should feature a reference constellation architecture as defined in section 3.2. That reference architecture is an inhomogeneous, symbiotic formation of about ten satellites, providing a regional service (tab. 3.3).

Not all characteristics of that reference architecture could be simulated right away. First of all, the available computing and emulation capacities limited the number of simulated satellites to four. Furthermore, the fact that only one verified on-board soft-

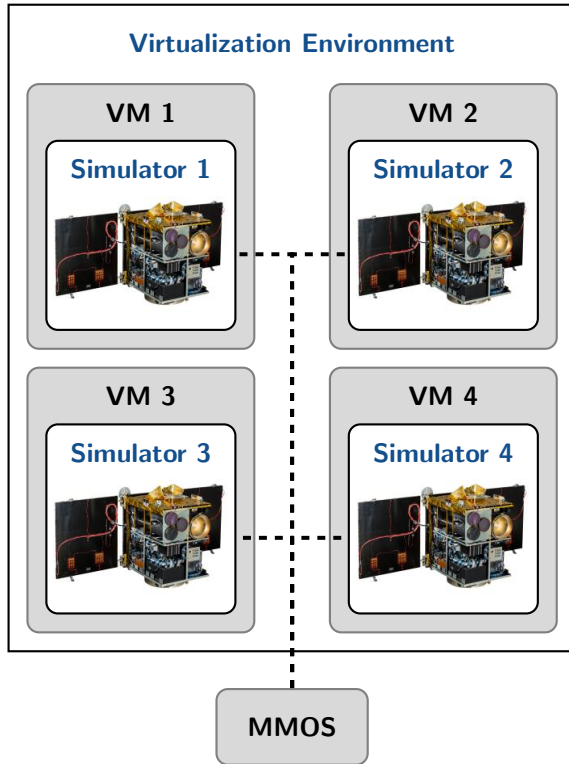


Figure 7.13: Simulated Example Mission

Table 7.2: *Flying Laptop* Orbit at 07 October 2021, 14:57:00, from [101]

Orbit Parameter		Value
Eccentricity	ϵ	0.001506
Inclination	i	97.4535°
Perigee Hight	h_p	583 km
Apogee Hight	h_a	604 km
RAAN	Ω	143.59°
Argument of Perigee	ω	131.07°
Mean Anomaly at Epoch	$\nu(\tau)$	229.18°
Epoch (UTC)	τ	06 October 2021, 19:21:58

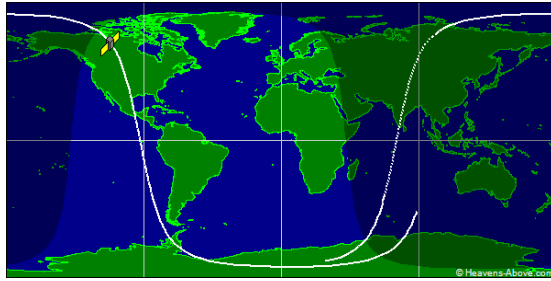
ware, and detail knowledge about only one satellite platform was available forced the simulation of four *identical* satellites (fig. 7.13).

7.3.1 Scenario

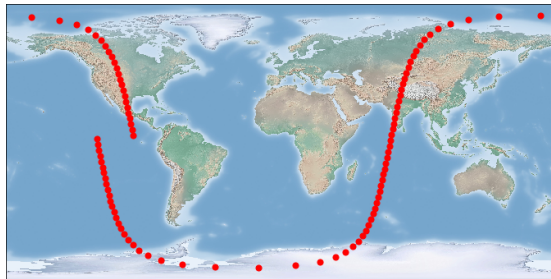
The selected scenario for the verification of the principle MMOS functionalities contains four simulated instances of *Flying Laptop* (fig. 7.13), each one executing a formally verified on-board software [9, 18]. All satellites are simulated in a separate orbit, similar to the one of the real mission [138]. *Flying Laptop* is operated in a sun-synchronous low Earth orbit (LEO). The orbital parameters from [101] are summarized in table 7.2. Plots of real and simulated ground tracks are displayed in figure 7.14.

To provide a realistic scenario, a fictive *Flying Laptop* constellation is simulated as it would suit the utilization of an Automatic Identification System (AIS). AIS is a system that allows tracking vessels. The reception of the signal emitted by the ship transmitters is normally limited to coastal regions. Installing AIS receivers within a satellite constellation would allow for a global coverage and the tracking of vessels in international waters.

For the simulation of such a constellation, four highly inclined, but otherwise identical orbits were selected, each one of course with



(a) Real Ground Track from [101]



(b) Simulated Ground Track

Figure 7.14: Plots of Real and Simulated Ground Tracks

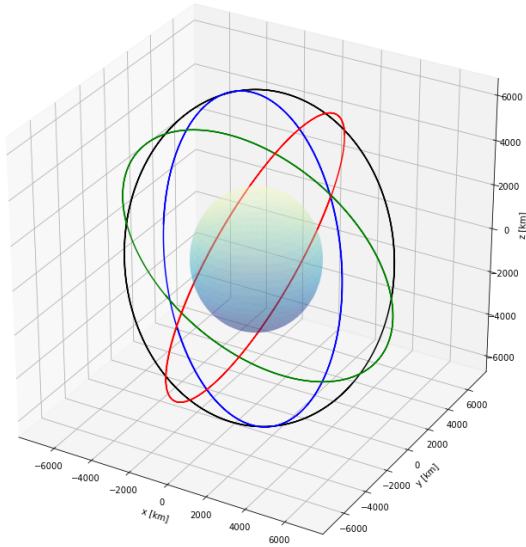
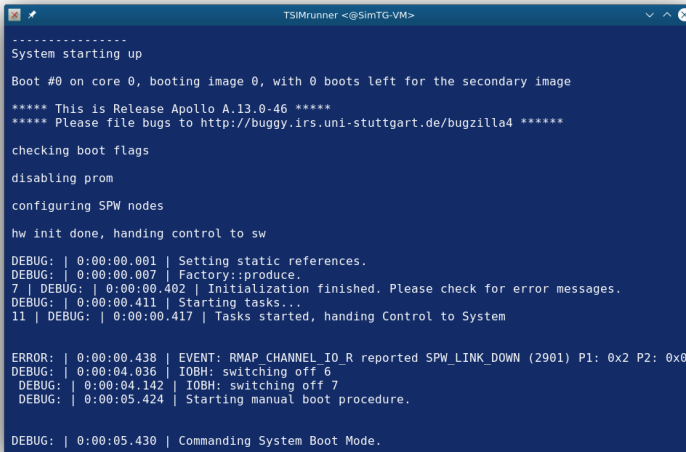


Figure 7.15: Simulated Orbits of a Fictive AIS Constellation based on *Flying Laptop* [138] - Orbits are plotted in the ECI frame. For the sake of clarity the Earth diameter has been reduced in size.



```
-----
System starting up

Boot #0 on core 0, booting image 0, with 0 boots left for the secondary image

***** This is Release Apollo A.13.0-46 *****
***** Please file bugs to http://buggy.irs.uni-stuttgart.de/bugzilla4 *****

checking boot flags

disabling prom

configuring SPW nodes

hw init done, handing control to sw

DEBUG: | 0:00:00.001 | Setting static references.
DEBUG: | 0:00:00.007 | Factory:produce.
7 | DEBUG: | 0:00:00.402 | Initialization finished. Please check for error messages.
DEBUG: | 0:00:00.411 | Starting tasks...
11 | DEBUG: | 0:00:00.417 | Tasks started, handing Control to System

ERROR: | 0:00:00.438 | EVENT: RMAP_CHANNEL_IO_R reported SPW_LINK_DOWN (2901) P1: 0x2 P2: 0x0
DEBUG: | 0:00:04.036 | IOBH: switching off 6
DEBUG: | 0:00:04.142 | IOBH: switching off 7
DEBUG: | 0:00:05.424 | Starting manual boot procedure.

DEBUG: | 0:00:05.430 | Commanding System Boot Mode.
```

Figure 7.16: Debugging Information provided by On-board Software Emulator

a separate argument of the ascending node (fig. 7.15). Three goals shall be achieved through the simulation of such a scenario.

1. Purpose of that simulator is to expose the MMOS to a realistic scenario, by leveraging the resources of and the knowledge from the *Flying Laptop* mission, available at the Institute of Space Systems.
2. In advance of the actual use case, these simulations further demonstrated that the selected computing hardware (tab. 7.1) is capable of executing four simulators in parallel without losing real-time performance [138].
3. Through the execution of simulation scenarios lasting several hours and days, the setup was further used to demonstrate the reliability of the simulator.

First runs of such a setup further demonstrated the proper integration of the OBC emulator into the simulator. A print of the integrated debug window, informing the user about the state of the hosted on-board software is shown in figure 7.16. Tests with a third-part CCS connected to the simulator (fig. 7.11) have shown, that emulated OBC and on-board software behave and process commands as expected.

7.3.2 Final Setup

As stated in the earlier course of this thesis, satellite operations is always the mutual interaction of three different systems: The OS, the ground station(s), and the satellite(s) (fig. 5.8). As a consequence, a space system is inhomogeneous and symbiotic by nature, even though the space segment is not. Therefore, the MMOS capabilities of operating inhomogeneous, symbiotic systems already emerge by its ability of planning, coordinating and executing space-to-ground interactions such as ground station passes (sec. 6.2.6.1).

To test the entire MMOS, not only the satellites need to be simulated, but the ground stations as well. A fully escalated setup is shown in figure 7.17. While satellites and MMOS exchange TM/TC via simulated SLE (or similar protocols), ground stations are commanded by means of a dedicated interface, which does not necessarily require an MCT (sec. 5.2.2.1). That antenna interface has not been addressed in the scope of this thesis though.

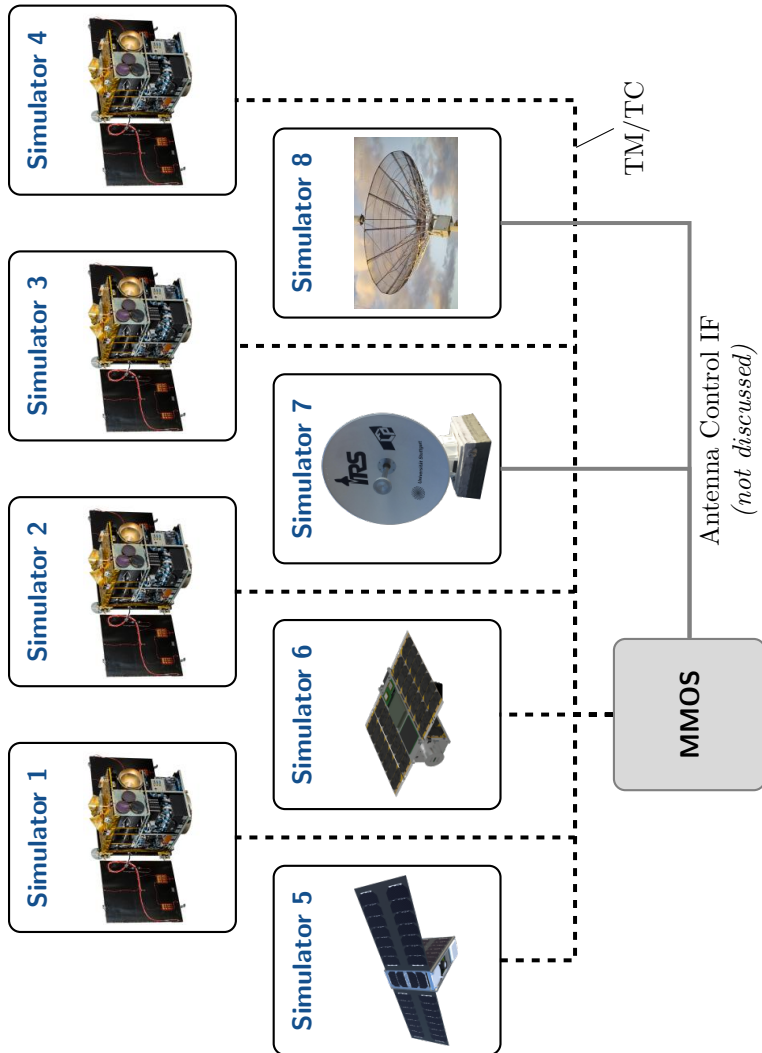


Figure 7.17: Fully Escalated Simulation Environment - Testing all MMOS functionalities requires the simulation of all operated systems, including ground stations. Antennas are commanded via dedicated control interfaces, which have not been discussed in the scope of this work. These control interfaces enable antenna planning and complement the SLE connection for the exchange of TM/TC with the satellites.

Summary

At the very beginning of this work, the term *complexity* was introduced. Constellations are complex by nature, due to the multitude of their satellite systems, and the amount of interactions between those and other systems in the operations process. A concept for constellation operations, or for the operations of multiple, individual satellites in parallel, must therefore be capable of handling this increased complexity. This work was conducted under the assumption that the most efficient way of dealing with a high level of system complexity is automation.

Result of this work is a macroscopic software architecture called the MMOS, designed for automatic operations of such satellite systems. Besides, the work focussed on the development of a resource-based planning concept within this architecture, which allows for the automatic generation, conflict resolution, execution, and verification of complex satellite activities.

Starting point of this development was a domain analysis, squired by an adaption towards modern satellite operation paradigms. Existing operations systems usually feature a central and very comprehensive mission control system carrying out more or less the entire

TM/TC based communication between ground and the operated system. These powerful mission control systems are then complemented by a bunch of further systems implementing the rest of the operational functionality. Such concepts are very prone to planing mistakes, as they do not actively support a lean automation process.

In the quest of defining an architecture that is an actual improvement compared to existing solutions, two aspects were considered important. First, efficient satellite operations requires a holistic view on the controlled assets, and as a direct consequence, satellite operations is a process that the developed architecture itself is a part of.

A holistic view on operations lead to the conclusion that a planning process needs to be implemented on system level, and that planning of complex systems requires taking into account interaction between different systems, respectively different types of systems. The implementation of a closed control process also requires the acquisition of detail knowledge about the state of these operated systems. Following existing research on automation and human-machine-interaction, an appropriate process was defined. That allowed for a detail specification of the MMOS subsystems, and the interfaces between those.

Furthermore, space and ground segment were decomposed into abstract communication layers. Through that kind of decomposition each interface enables a ground component interacting with a counter part in space. Outcome of this development was a system layer interface, called *Activity*.

Its particular value for the planning process relies on two characteristics. First, Activities can be nested, which means that one Activity can be decomposed into several child Activities. This way,

a task can not just be split into subtasks, they can also be assigned to various systems, which allows for a planning process across system boundaries, and thus the planning of system interaction. Second, Activities specify a resource demand. In combination with a respective resource modelling approach, this allows for the prediction of the system state based on a given schedule, as well as for an automatic conflict resolution.

By means of the results summarized so far, the first research hypothesis (sec. 1.3) formulated for this work can be answered positively, because all theoretical requirements for the automatic operations of a remote satellite are met and supported by the introduced methodology.

The component within the MMOS implementing the scheduling algorithms, the resource propagation, and the conflict resolution is the MPT. Through the implementation of a state machine within its scheduling layer, each MPT instance has the sovereign authority on the Activity schedule of an operated system. The nesting quality of Activities allows for the clustering of multiple MPTs and thus for the representation of complex satellite systems like constellations on ground.

This quality of the MPS supports the second research hypothesis (sec. 1.3). The proposed design supports the operations of a finite number of individual satellites, based on paradigms for automatic operations of a single system. However, it turned out, that for the automatic operations of a complex system, like a constellation, it is never sufficient to just look at a single satellite.

While former operations system architectures used to be primarily data driven, the proposed approach is process driven. The interaction with the spacecraft is further no longer (just) based on the

exchange of TM/TC, but on the scheduling of Activities. The fact that these schedules are pre-checked and verified, and an activity schedule comes along with a forecast of the system state, allows for complete new approaches for spacecraft application development on ground. So, individual, mission specific applications can be developed with pure focus on their mission goal and less focus on the validity of the resulting system schedule, as the latter is carried out by the introduced concept. The interface for the integration of such applications into the MMOS is again the *Activity*.

Outlook

At submission of this thesis, the first versions of the MMOS have been released featuring an integrated telecommand chain that consists of MPT, MCT, and VGS. That command chain implements the basic functionalities such as Activity state management, automatic command release and verification, and a protocol conversion by the MCT.

By means of a GUI, operators can interact with the mission schedule, managed by the MPT. It allows for the addition of Activities to the schedule, supports the modification of Activities, and allows for the monitoring of the Activity states. Furthermore, the user interface supports the manual release of commands, and gives access to the complete history of sent commands.

An integrated user management further limits and controls the access to the MMOS microservices underneath, and ensures for example, that only authorized personnel can trigger a command release or modify the schedule.

Based on this implementation, the next logical steps are the following.

- The integration of Flight Dynamics and Resource Propagation as described is a missing step towards a complete MPT. The propagation of the system state and the propagation of the

orbit are the basis for the conflict management by the MPT. The MPT further needs to be connected with the DAS for the retrieval of satellite telemetry from the archive. All the necessary interfaces were introduced and defined in this thesis.

- After the development and the integration of the remaining components stands the formal verification of the entire system. This covers the checkout of the entire communication chain, beginning with the scheduling of Activities, the release and the transmit of commands, the verification and the display of telemetry, and the closing of the Activities eventually. An appropriate verification infrastructure on the basis of a fully simulated satellite has been set up in the course of this work.
- The next step would be the orchestration of an MMOS featuring multiple instances of interacting system MPTs, aiming for the verification that the concept of nested Activities has been implemented properly. The verification infrastructure featuring multiple simulated satellites has been implemented and is ready for appropriate test scenarios.
- The mission planning concept introduced in this work is agnostic of any kind of operated system. A foreseen application is the use of MPTs for the scheduling of ground stations. Thus, the next logical step would be a connection of the MMOS to a ground station infrastructure and to perform analyses identifying to what degree the specified interfaces support the monitoring and command of antenna systems.
- To date a number of new satellite projects at the University of Stuttgart entered the integration phase and are to be launched within months. With the first version of the MMOS in place, the development teams can start checking out their systems and start building their own Applications (Agents) on top of the MMOS.

- The biggest outcome of this work is the Activity based scheduling concept. The design of the MMOS, featuring such an integrated scheduling mechanism, together with a standardized interface to that schedule open a lot of different research opportunities, and possibilities to extend the functional scope of the system. It further allows widening the user group of the MMOS. Until the development of the MMOS access to the OS, and thus access to the satellite, was limited to the operations team and some few stakeholders like engineers or mission scientists. With such a system in place various kinds of applications can be designed on top of the MMOS. Such applications could be used to provide a larger scientific community with access to satellites and satellite data. Appropriate research projects have already been engaged.

*

Information Used

A.1 List of Satellite Constellations

Table A.1: List of completed and currently developed satellite constellations sorted by launch year of the first satellite. Compiled by [11]

Name	Organization	Region	Year	Active	Total	App
GPS	USAF	USA	1978	31	72	Nav.
GLONASS	Roscosmos	RUS	1982	24	26	Nav.
Feng Yun	National Satellite Me- teorological Centre	CHN	1988	9	17	EO
Beidou	CNSA	CHN	2000	33	35	Nav.
Cluster II	ESA	EU	2000	4	4	Sci.
AprizeSat	AprizeSat	USA, CAN	2002	12	12	Com.

Table continues on next page.

Table A.1: List of satellite constellations. Compiled by [11]

Name	Organization	Region	Year	Active	Total	App
DMCii	DMC International Imaging	Int.	2002	7	13	EO
Meteosat	ESA	EU	2002	4	10	EO
A-Train	Various	Int.	2002	7	7	EO
SaudiComSat	KACST	SAU	2004	7	24	Com.
Gonets D1M	RKA	RUS	2005	12	27	Com.
Formosat 3 / COSMIC	NSPO, NOAA	TWN, USA	2006	6	6	EO
MetOp	ESA	EU	2006	2	9	EO
CartoSat	ISRO	IND	2007	7	10	EO
COSMO-Skyne	ASI	ITA	2007	4	4	EO
RapidEye	Planet Labs, Inc.	USA	2008	5	5	EO
Huan-Jing	CRESDA	CHN	2008	3	3	EO
Globalstar 2nd gen.	Globalstar	USA	2010	24	24	Com.
QZSS Michibiki	JAXA	JPN	2010	4	4	Nav.
Pleiades SPOT	Astrium	FRA	2011	4	4	EO
Orbcomm-OG2	Orbcomm	USA	2012	12	18	Com.
FireBIRD	DLR	GER	2012	2	2	EO
O3b	O3b Networks Ltd.	USA	2013	20	42	Com.

Table continues on next page.

Table A.1: List of satellite constellations. Compiled by [11]

Name	Organization	Region	Year	Active	Total	App
BRITE	Various	CAN, POL, AUT	2013	5	5	Sci.
Gafon CHEOS	CNSA	CHN	2013	12	12	EO
IRNSS	ISRO	IND	2013	(9)	7	Nav.
SWARM Earth Explorer	ESA, CSA	EU, CAN	2013	4	4	Sci.
Flock	Planet Labs, Inc.	USA	2014	188	300	EO
Sentinel	ESA	EU	2014	7	14	EO
Lemur-2	Spire Global Inc.	USA	2015	100	175	EO
Galileo	ESA	EU	2015	22	30	Nav.
Jilin	Chang Guang Satellite Technology Co.	CHN	2015	13	138	Data
MMS	NASA, SwRI	USA	2015	4	4	Sci.
SkySat	Planet Labs, Inc.	USA	2016	15	21	EO
GHGSat	GHGSat Inc.	CAN	2016	1	3	EO
NuSat	Satellogic S.A.	ARG	2016	7	25	EO
GaoJing SuperView	Siwei Star Company	CHN	2016	4	16	EO
CYGNSS	NASA	USA	2016	8	8	EO
Global	BlackSky	USA	2016	4	60	EO

Table continues on next page.

Table A.1: List of satellite constellations. Compiled by [11]

Name	Organization	Region	Year	Active	Total	App
Landmapper BC	Astro Digital	USA	2017	4	10	EO
3 Diamonds / Perls	Sky and Space Global	GBR, ISR, AUS	2017	3	8	Com.
CICERO	GeoOptics Inc.	USA	2017	7	24	EO
Iridium Next	Thales Alenia, Orbital ATK	USA	2017	74	81	Com.
OneWeb	OneWeb	USA	2018	6	648	Com.
Starlink	SpaceX	USA	2018	0	4425	Com.
Starlink (VLEO)	SpaceX	USA	2018	240	9102	Com.
Telesat	Telesat Canada	CAN	2018	1	292	Com.
Kepler	Kepler Communications Inc.	CAN	2018	1	140	Data
Axelglobe	Axelspace	JPN	2018	3	50	EO
Commstallation	Microsat Systems Canada Inc.	CAN	2018	0	84	Com.
n.n.	Hiber	NLD	2018	2	50	IoT
Astrocast	Astrocast	CHE	2018	2	64	IoT
Landmapper HD	Astro Digital	USA	2018	0	20	EO
PlatentiQ	PlanetiQ	USA	2018	0	12	EO
n.n.	Capella Space	USA	2018	1	36	EO

Table continues on next page.

Table A.1: List of satellite constellations. Compiled by [11]

Name	Organization	Region	Year	Active	Total	App
n.n.	Fleet Space Technologies	USA	2018	4	100	IoT
n.n.	Reaktor Space Lab	FIN	2018	1	36	EO
n.n.	SatRevolution	POL	2018	2	1024	EO
n.n.	AISTech Space	ESP	2018	2	102	EO
GEMS	Orbital Micro Systems	GBR	2018	1	48	EO
n.n.	HawkEye 360 Inc.	USA	2018	3	18	EO
SeaHawk	University of North Carolina	USA	2018	1	10	EO
ICEYE	ICEYE	FIN	2018	4	18	EO
n.n.	Astronome Technologies	IND	2019	0	150	Com.
1HOPSat	Hera Systems	USA	2019	1	50	EO
n.n.	NSLComm	ISR	2019	1	60	Com.
n.n.	Kleos Space	LUX	2020	0	20	EO
n.n.	Lacuna Systems	GBR	2021	0	32	IoT
SpaceBelt	Cloud Constellation Corporation	USA	2021	0	12	Data
ViaSat	ViaSat Inc.	USA	2021	0	20	Com.
UrtheDaily	UrtheCast	CAN	2022	0	8	EO

Table continues on next page.

Table A.1: List of satellite constellations. Compiled by [11]

Name	Organization	Region	Year	Active	Total	App
Space Norway	Space Norway AS	NOR	2022	0	2	Com.

A.2 The LOAT Matrix

Table A.2: Complete LOAT Matrix, as defined by Save and Feuerberg
- from [113]

A	B	C	D
Information Acquisition	Information Analysis	Decision and Action Selection	Action Implemen- tation
A0	B0	C0	D0
The human ac- quires relevant information on the process she/he is fol- lowing without using any tool.	The human compares, combines and analyses different infor- mation items regarding the status of the process she/he is following by way of mental elaborations. She/he does not use any tool or support external to her/his work- ing memory.	The human generates de- cision options, selects the ap- propriate ones and decides all actions to be performed.	The human ex- ecutes and con- trols all actions manually.

Table continues on next page.

Table A.2: Complete LOAT Matrix, as defined by Save and Feuerberg
- from [113]

Information Acquisition	Information Analysis	Decision and Action Selection	Action Implementation
A1	B1	C1	D1
The human acquires relevant information on the process she/he is following with the support of low-tech non-digital artefacts.	The human compares, combines, and analyses different information items regarding the status of the process she/he is following utilising paper or other non-digital artefacts.	The human generates decision options, selects the appropriate ones and decides all actions to be performed utilising paper or other non-digital artefacts.	The human executes and controls actions with the help of mechanical non-software based tools.

Table continues on next page.

Table A.2: Complete LOAT Matrix, as defined by Save and Feuerberg
- from [113]

Information Acquisition	Information Analysis	Decision and Action Selection	Action Implementation
A2	B2	C2	D2
The system supports the human in acquiring information on the process she/he is following. Filtering and/or highlighting of the most relevant information are up to the human.	Based on user's request, the system helps the human in comparing, combining and analysing different information items regarding the status of the process being followed.	The system proposes one or more decision alternatives to the human, leaving freedom to the human to generate alternative options. The human can select one of the alternatives proposed by the system or her/his own one.	The system assists the operator in performing actions by executing part of the action and/or by providing guidance for its execution. However, each action is executed based on human initiative and the human keeps full control of its execution.

Table continues on next page.

Table A.2: Complete LOAT Matrix, as defined by Save and Feuerberg
- from [113]

Information Acquisition	Information Analysis	Decision and Action Selection	Action Implementation
A3	B3	C3	D3
The system supports the human in acquiring information on the process she/he is following. It helps the human in integrating the data coming from different sources and in filtering and/or highlighting the most relevant information items, based on user's settings.	Based on user's request, the system helps the human in comparing, combining and analysing different information items regarding the status of the process being followed. The system triggers visual and/or aural alerts if the analysis produces results requiring attention by the user.	The system proposes one or more decision alternatives to the human. The human can only select one of the alternatives or ask the system to generate new options.	The system performs automatically a sequence of actions after activation by the human. The human maintains full control of the sequence and can modify or interrupt the sequence during its execution.

Table continues on next page.

Table A.2: Complete LOAT Matrix, as defined by Save and Feuerberg
- from [113]

Information Acquisition	Information Analysis	Decision and Action Selection	Action Implementation
A4	B4	C4	D4
The system supports the human in acquiring information on the process she/he is following. The system integrates data coming from different sources and filters and/or highlights the information items which are considered relevant for the user. The criteria for integrating, filtering and highlighting the relevant information ...	The system helps the human in comparing, combining and analysing different information items regarding the status of the process being followed, based on parameters pre-defined by the user. The system triggers visual and/or aural alerts if the analysis produces results requiring attention by the user.	The system generates options and decides autonomously on the actions to be performed. The human is informed of its decision.	The system performs automatically a sequence of actions after activation by the human. The human can monitor all the sequence and can interrupt it during its execution.

Table continues on next page.

Table A.2: Complete LOAT Matrix, as defined by Save and Feuerberg
 - from [113]

Information Acquisition	Information Analysis	Decision and Action Selection	Action Implementation
--------------------------------	-----------------------------	--------------------------------------	------------------------------

are predefined at design level but visible to the user.

A5	B5	C5	D5
The system supports the human in acquiring info on the process s/he is following. The system integrates data coming from different sources and filters and/or highlights the information items considered relevant for the user.	The system performs comparisons and analyses of data available on the status of the process being followed based on parameters defined at design level. The system triggers visual and/or aural alerts if the analysis produces results requiring attention by the user.	The system generates options and decides autonomously on the action to be performed. The human is informed of its decision only on request.	The system initiates and executes automatically a sequence of actions. The human can monitor all the sequence and can modify or interrupt it during its execution.

Table continues on next page.

Table A.2: Complete LOAT Matrix, as defined by Save and Feuerberg - from [113]

Information Acquisition	Information Analysis	Decision and Action Selection	Action Implementation
		<p>C6</p> <p>The system generates options and decides automatically on the action to be performed without informing the human. (Always connected to an Action Implementation level not lower than D5.)</p>	<p>D6</p> <p>The system initiates and executes automatically a sequence of actions. The human can monitor all the sequence and can interrupt it during its execution.</p>

Table continues on next page.

Table A.2: Complete LOAT Matrix, as defined by Save and Feuerberg
- from [113]

Information Acquisition	Information Analysis	Decision and Action Selection	Action Implemen- tation
			<p>D7</p> <p>The system initiates and executes a sequence of actions. The human can only monitor part of it and has limited opportunities to interrupt it.</p>
			<p>D8</p> <p>The system initiates and executes a sequence of actions. The human cannot monitor nor interrupt it until the sequence is not terminated.</p>

B

MPT Implementation

B.1 MPT Requirement Backlog

Table B.1: MPT Requirement Backlog with Prioritization Scheme - The backlog is according to the functional architecture in section 6.3

Name	Description	Prio.	Ver.
Communication Layer	The MPT shall have a low level communication layer (COMM) by means of which the MPT is connected to the rest of the system, and which handles all the traffic between the MPT and other system components.	Must	Rev.
Interface Layer	The MPT shall have an interface layer (IFL). Purpose of the IFL is to enable functional components of the MPT collaborating with other system functions.	Must	Rev.

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
Operative Layer	The MPT shall have an operative layer (OPL). Purpose of the OPL is to perform various actions based on the Activity states.	Must	Rev.
Scheduling Layer	The MPT shall have a scheduling layer (SCL). Purpose of the SCL is to host the services, which manage the state of the scheduled activities.	Must	Rev.
Configuration Layer	The MPT shall have configuration (CONF) layer. Purpose of the CONF layer is to initialize the MPT, to load the mission specific configuration of the tool and to gather information about the status of the tool.	Should	Rev.
Gateway	The communication layer shall implement a gateway to connect to the Middleware	Must	Insp.
Routing	The Gateway shall be capable of directing messages to a specific component in the System	Must	Test
Reception	The Gateway shall be capable or recognizing messages from a different component in the system and provide the content to the respective feature in the MPT.	Must	Test

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
Downtime	After a down-time of the MPT, the gateway shall be capable of collecting messages, buffered by the Middleware and provide them the target features in the MPT.	Should	Test
Activity & Schedule Interface	The MPT shall have an interface to access the mission schedule, which is stored in a central database. That IF shall be called the Activity & Schedule Interface (ASI)	Must	Insp.
Phase Interface	The MPT shall have an interface to make phase requests at the FDT. That IF shall be called the Phase Interface (PIF)	Should	Insp.
Data Interface	The MPT shall have an interface to connect to the central system TM Archive. That IF shall be called the Data Interface (DIF)	Should	Insp.
Monitoring Interface	The MPT shall have an interface that allows other (system management) instances to monitor the state of the MPT as an instance. That IF shall be called the Monitoring Interface (MIF)	Could	Insp.

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
Control Inter- face	The MPT shall have an interface that allows other (system management) instances to control the MPT as an instance. That IF shall be called the Control Interface (CIF)	Could	Insp.
MIB Interface	The MPT shall have an interface that allows mission specific configurations from the MIB into the MPT. That interface shall be called MIB Interface (MIBIF)	Won't	Insp.
Buffering	The interface layer shall deal with the fact that requests to other system components won't return right away. It shall buffer the returned information so that the MPT component making the request can collect the requested data later on.	Must	Test
Schedule Queries	The ASI shall allow making queries within the mission schedule and return the requested information	Must	Test
Activity Mod- ification	The ASI shall allow modifying activities within the mission schedule and return information about the success of the modification	Must	Test

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
Activity Information	The ASI shall allow for getting information about a specific activity in the schedule	Must	Test
Command Modification	The ASI shall allow to make modifications on the scheduled commands and return information about the success of the modification.	Must	Test
Command Information	The ASI shall allow for getting information about a specific command in the schedule.	Must	Test
Multiple Mission Schedules	The ASI must be capable of accessing multiple mission schedules	Must	Test
Return of Phase Information	The PIF shall enable components of the MPT to request phases from a respective component in the system.	Must	Test
Orbit Information Request	The PIF shall allow for the request of orbit information for a specific time.	Should	Test
Telemetry Information	The DIF shall enable components of the MPT querying the system telemetry from a central archive.	Must	Test
TM request	The DIF shall allow for the request of one or more parameters for a specified time frame	Must	Test
Raw Value	The DIF only returns the raw parameter values.	Must	Test

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
MPT active	The MIF shall allow to follow if the MPT as a component is working	Must	Test
Functions Monitoring	The MIF shall allow to follow if each functional component (services, management components, threats) are active.	Must	Test
Statistics	The MIF shall allow to monitor statistics (such as data throughput, handled items per duration, amount of handled items, active since ... information, etc.) the data of interest of course depends on the respective function.	Could	Test
MPT Component State	Each MPT component shall be able to be monitored. Monitored is the component state (active, paused, terminated, stuck, etc...)	Should	Test
Statistics Generation	Each component shall be capable of gathering relevant statistics. The gathered data of interest depends on the respective component functionality (e.g data throughput, handled items, load, ...)	Should	Test

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
MPT Component Control	The CIF shall enable other components to control (start, stop, pause and reset) functional components (services, management components) within the MPT.	Must	Test
MPT Function Calls	The CIF shall allow for specific function calls.	Won't	Test
Configuration During Runtime	The CIF shall allow for an MPT configuration during runtime.	Should	Test
MPT Component Configuration	Each MPT functional component shall be configurable. The extend of the configuration depends on the component functionality.	Should	Insp.
MPT Runtime Configuration	Each MPT component shall be configurable during runtime	Should	Test
CMD Management	The MPT shall have a command management	Must	Insp.
Release Management	The MPT shall have release management	Must	Insp.
Resource Management	The MPT shall have a resource management	Should	Insp.
On-board Schedule Mgmt.	The MPT shall have an on-board schedule management	Won't	Insp.
Orbit Management	The MPT shall have an orbit management	Won't	Insp.

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
IFL Latencies	The operative layer shall be designed in a way, so it can handle the fact that requests via the underlying IFL won't return right away.	Should	Test
Release Unlock	A releaser in the CMD Management shall unlock commands in the schedule for release	Must	Test
Unlock Condition	Only commands of activities in state 03 (in transmission) shall be unlocked for release	Must	Test
Lock Condition	Unlocked CMDs of activities which are not in state 03 shall be locked.	Must	Test
Command Ownership	The CMD management shall only modify commands which are part of an activity managed by this MPT.	Must	Test
Release Time	The release management shall determine the CMD release times based on the CMD execution times, scheduled uplinks, and activity status.	Must	Test
Release Planning	Release times shall only be determined for commands of activities in state 02 or 11.	Must	Test
Pass select	The release planning must be capable of querying (up)link activities from the schedule via the ASI.	Must	Test

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
Pass duration	The release planning must be capable of querying querying ground station passes via the PIF.	Must	Test
Link duration	The duration of a link results from the duration of the ground station pass and the scheduled link activity. Note: The link duration is NOT equal to the pass duration!	Must	Test
Link time margin	The release planning shall consider configurable time margins at the beginning and the end of the link	Must	Test
Data volume, bandwidth, and duration	The release planning shall consider the uplink data volume, the bandwidth and the determined link duration for the release planning.	Should	Test
Release Rules	Different rules can be applied for the determination of the release times.	Could	Insp.
Release Rule Config	The release planning rules can be configured	Could	Insp.
Select of Release Rules	The release planning rules can be selected by the user	Could	Insp.

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
Release Optimization	The determined release times are the result of an optimization. The variable to be optimized is a matter of configuration or a user input.	Won't	Test
Resource	A resource is a mapped system parameter, which is considered as part of the system state.	Must	Rev.
Accountant	The resource management shall manage budgets for each consumed resource. The software component in charge of the budgets shall be called the Accountant	Must	Insp.
Budget	A budget tracks by whom (in course of which activity) a resource is consumed and how much.	Must	Test
Budget Timeline	A budget tracks the state of a resource in time.	Must	Test
Resource Propagation	The resource management shall propagate the state of the resources based on their last known state and based on the scheduled activities consuming them.	Must	Test
Demand Handling	For the propagation of the resource state, the resource management shall handle demands, specified by the activities	Must	Test

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
Absolute Demand	The resource management shall be capable of handling absolute demands: Absolute demands specify a specific state the resource shall be in (e.g. system mode = 'safe')	Must	Test
Relative Demand	The resource management shall be capable of handling relative demands: Relative demands specify how much of a resource is consumed during an activity (e.g. 2 MB of RAM)	Must	Test
Multiple Demands	An activity can specify multiple demands	Must	Test
Resource Information	The resource management informs about the scheduled state of a resource at a given time	Should	Test
Resource Check	The resource management shall handle requests for an activity resource check. In case of a conflict it informs the requester about the cause of the conflict and the conflicting activities	Should	Test
On-board Schedule Mgmt.	The OBS management shall keep track of the on-board schedules.	Must	Test
OBS Functionality	The functional extend of the OBS management is still to be defined.	Must	Test

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
Orbit	An orbit is a combination of multiple parameters describing the shape and the location of the satellite trajectory at a specific time.	Must	Rev.
Orbit Management Implementation	Due to its similar functionality, the orbit management should be derived directly from the resource management. Instead of a single resource parameter, managing orbits requires the propagation of multiple parameters (e.g. six Keplerian Elements)	Should	Ana.
Navigator	The orbit management keeps track of the satellite's trajectory (aka. The scheduled future orbits). The software component in charge of the orbits shall be called Navigator	Must	Test
Orbit Schedule	The orbit schedule keeps of the maneuver activities and the target orbits.	Must	Test
Orbit Timeline	The orbit schedule keeps track or the trajectory in time (scheduled orbits in time)	Must	Test

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
Orbit Propagation	The orbit management shall propagate the trajectory based on the last known orbit and based on the scheduled maneuver activities. This requirement might require an interaction with the FDT (in case this functionality is not implemented within the orbit management itself).	Must	Test
Maneuver Data	The orbit management shall be capable of handling a defined set of maneuver data. That set of data is to be defined.	Must	Test
Orbit Information	The orbit management informs about the scheduled orbit at a given time	Should	Test
Orbit Check	The orbit management shall handle requests for maneuver checks. In case of a conflict (maneuver does not fit into the scheduled trajectory), the orbit management informs the requester about the cause of the conflict and the conflicting maneuver activities	Should	Test
19 Threads	The SCL shall have a dedicated thread for each of the 19 possible states of an activity	Must	Insp.
Thread Basic Function	Purpose of a thread is to transform an activity from a specific state into another state.	Must	Test

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
IFL and OPL latencies	The thread layer shall be designed in a way, so it can handle the fact that requests via the underlying layers (IFL and OFL) won't return right away.	Must	Test
Schedule Modification	The SCL shall be capable of writing the mission schedule by means of the ASI.	Must	Test
Service Thread	Each thread is a continuous thread constantly checking and updating the state of its activities.	Must	Test
Only State Modification	The Service only changes the state of an activity. Any effects caused through the change of an activity status are implemented by other components.	Must	Test
Activity State Machine	The SCL shall implement a state machine that results in a state flow as specified in the activity definition. An activity must not change into a state that is not foreseen in the concept. (e.g. an activity must not change from state 02 into state 06.)	Must	Test
Gathering Information	The SCL shall be able to gather all information required to check the activity states.	Must	Test

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
Activity Information	The SCL shall be capable to request information about an activity and its attributes from the mission schedule via the ASI.	Must	Test
Command Information	The SCL shall be capable to request information about the commands within the activity. Such request shall be made at the mission schedule via the ASI.	Must	Test
Real System Data	The SCL shall be able to gather system parameters from the central archive via the DIF.	Should	Test
Resource Schedule Data	The SCL shall be able to gather the scheduled state of a resource from the resource management.	Should	Test
Resource Verification	The SCL shall be capable of comparing real satellite data with the scheduled resource state in order to verify the successful execution of an activity.	Could	Test
Orbit Verification	Like a system resource, the SCL must be capable of comparing the real orbit with the scheduled orbit.	Won't	Test
Thread 01	Thread 01 shall handle activities in state 01 and check conditions based on which the activity is either suspended, set failed or set to state 02	Must	Test

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
Thread 02	Thread 02 shall handle activities in state 02 and check conditions based on which the activity is either suspended, set failed or set to state 03	Must	Test
Thread 03	Thread 03 shall handle activities in state 03 and check conditions based on which the activity is either suspended, set failed or set to state 04	Must	Test
Thread 04	Thread 04 shall handle activities in state 04 and check conditions based on which the activity is either suspended, set failed or set to state 05	Must	Test
Thread 05	Thread 05 shall handle activities in state 05 and check conditions based on which the activity is either suspended, set failed or set to state 06	Must	Test
Thread 06	Thread 06 shall handle activities in state 06 and check conditions based on which the activity is either set failed or set to state 07	Must	Test
Thread 07	Thread 07 shall handle activities in state 07. Purpose of Thread 07 is to close the activity eventually	Must	Test
Thread 08	Thread that handles all closed activities. This thread possibly has no functionality.	Must	Test

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
Thread 09	Thread 09 shall handle activities in state 09 and check conditions based on which the activity is either resumed, or set failed.	Should	Test
Thread 10	Thread 10 shall handle activities in state 10 and check conditions based on which the activity is either resumed, or set failed.	Should	Test
Thread 11	Thread 11 shall handle activities in state 11 and check conditions based on which the activity is either resumed, or set failed.	Should	Test
Thread 12	Thread 12 shall handle activities in state 12 and check conditions based on which the activity is either resumed, or set failed.	Could	Test
Thread 13	Thread 13 shall handle activities in state 13 and check conditions based on which the activity is either resumed, or set failed.	Won't	Test
Thread 14	Thread 14 shall handle activities in state 14. Purpose the thread is to close the activity	Should	Test
Thread 15	Thread 15 shall handle activities in state 15. Purpose the thread is to close the activity	Should	Test
Thread 16	Thread 16 shall handle activities in state 16. Purpose the thread is to close the activity	Should	Test

Table continues on next page.

Table B.1: MPT Requirement Backlog with Prioritization Scheme

Name	Description	Prio.	Ver.
Thread 17	Thread 17 shall handle activities in state 17. Purpose the thread is to close the activity	Could	Test
Thread 18	Thread 18 shall handle activities in state 18. Purpose the thread is to close the activity	Could	Test
Thread 19	Thread 19 shall handle activities in state 19. Purpose the thread is to close the activity	Won't	Test
Monitoring & Control	The CONF layer shall implement and execute an internal MPT monitoring & control	Could	Insp.
Configuration	The CONF layer shall implement means of configuring the MPT for a specific system	Should	Insp.

B.2 MPT Activity Management Flow Diagrams

In section 6.3 the MPT architecture was introduced. The following compilation of figures shows the state machines as implemented in the nineteen different threads of the MPT scheduling layer.

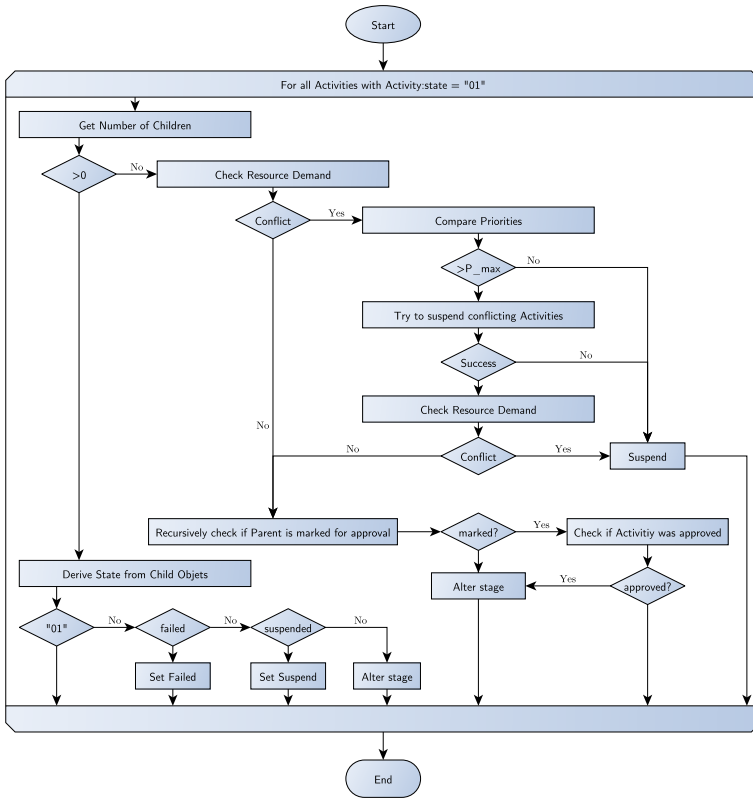


Figure B.1: Thread 1 State Machine

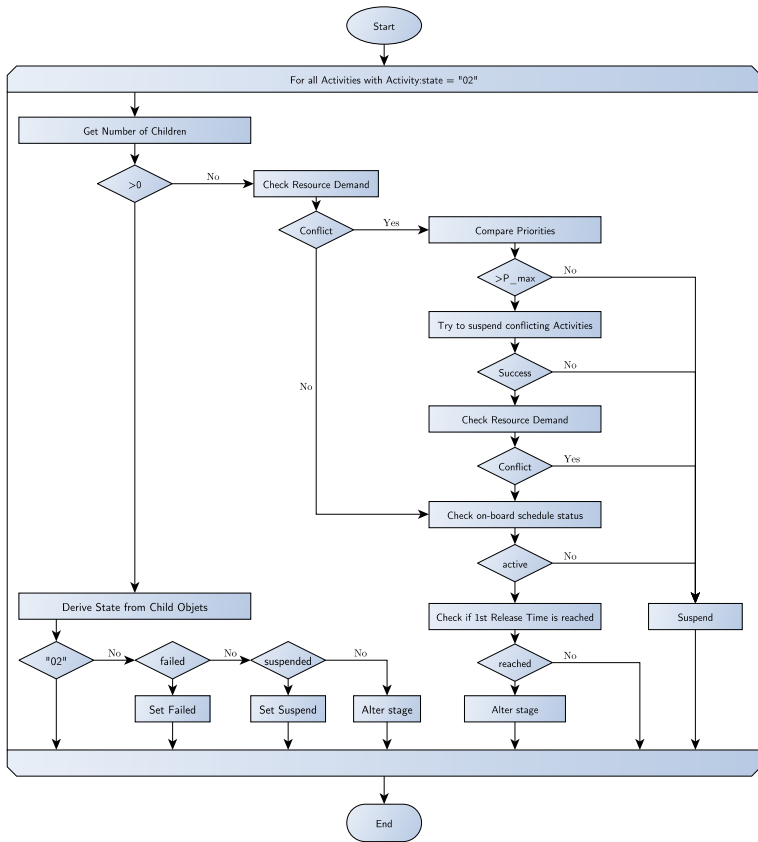


Figure B.2: Thread 2 State Machine

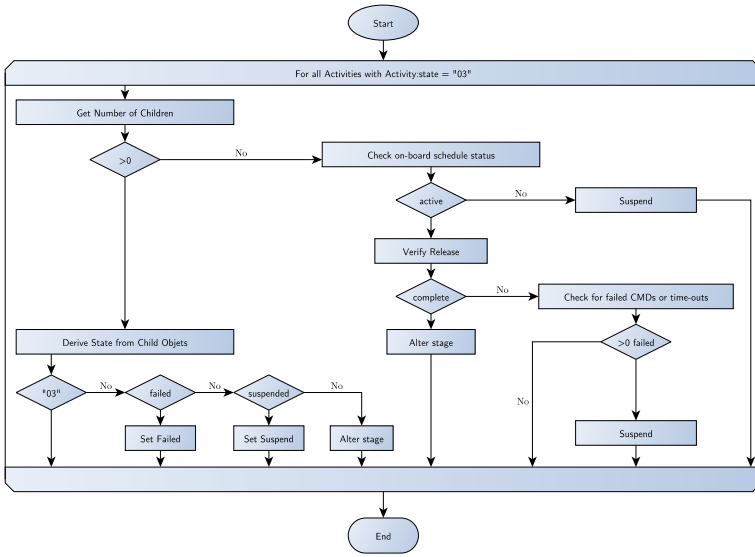


Figure B.3: Thread 3 State Machine

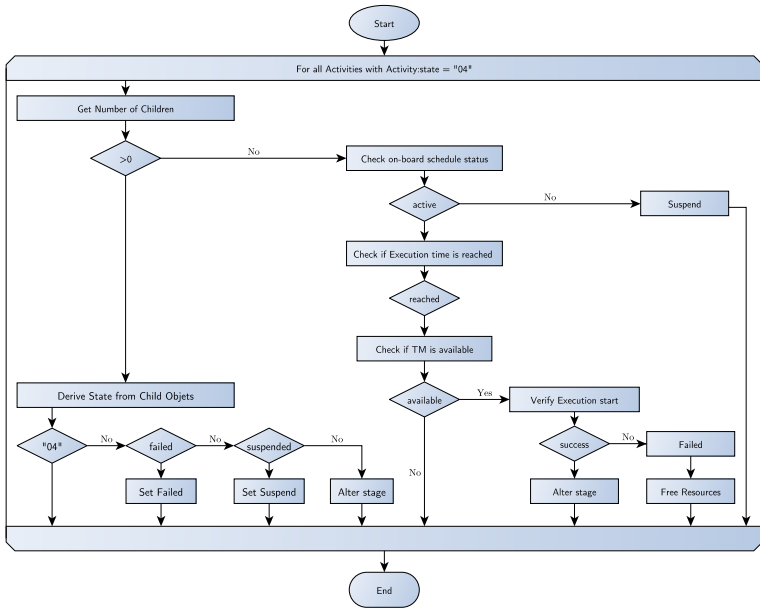


Figure B.4: Thread 4 State Machine

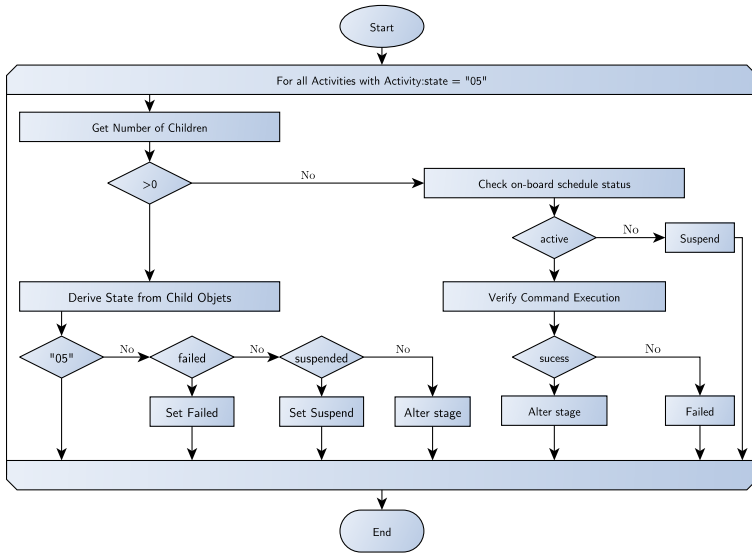


Figure B.5: Thread 5 State Machine

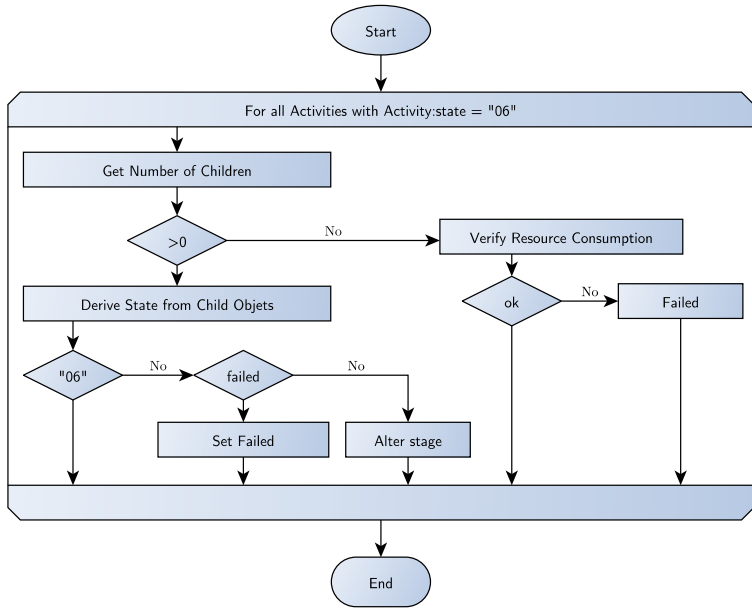


Figure B.6: Thread 6 State Machine

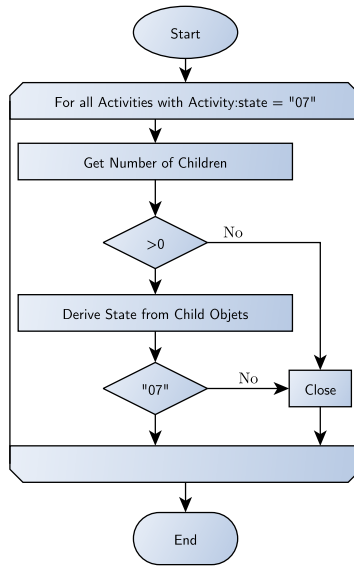


Figure B.7: Thread 7 State Machine

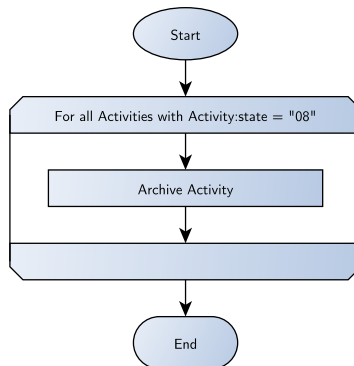


Figure B.8: Thread 8 State Machine

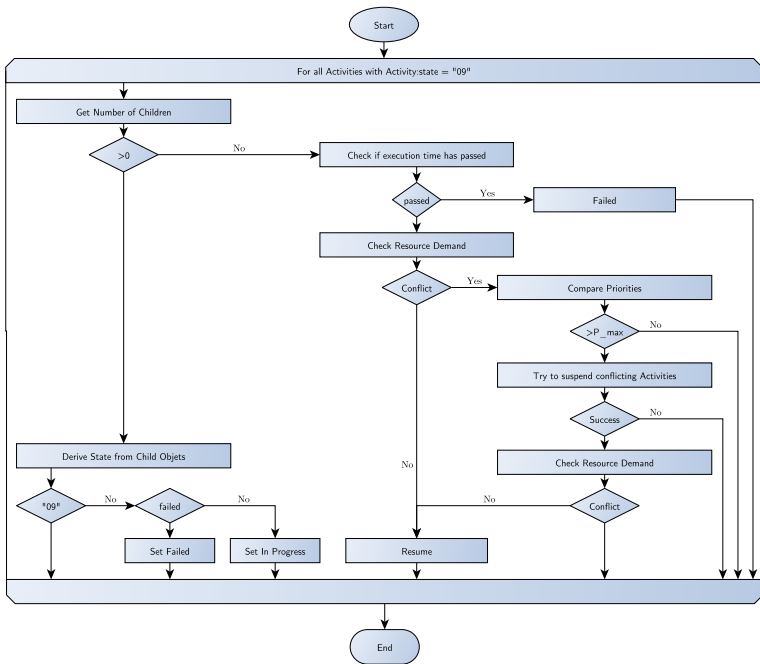


Figure B.9: Thread 9 State Machine

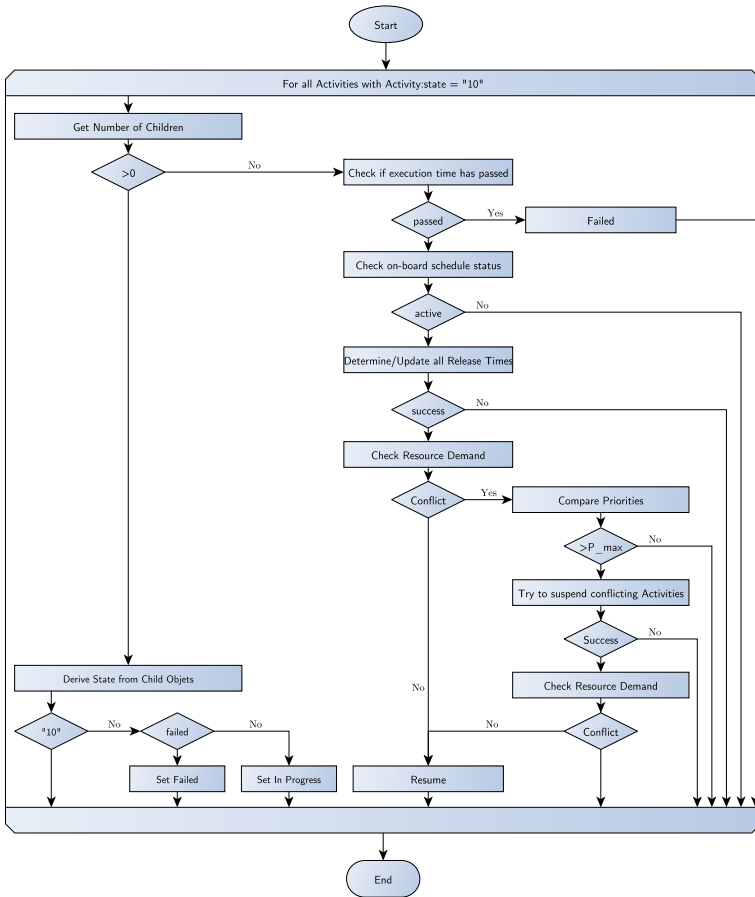


Figure B.10: Thread 10 State Machine

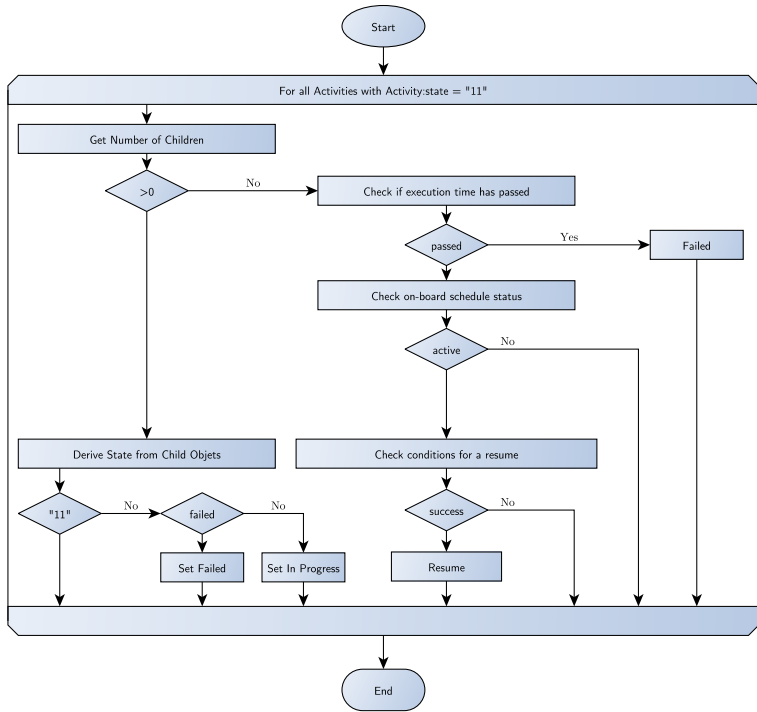


Figure B.11: Thread 11 State Machine

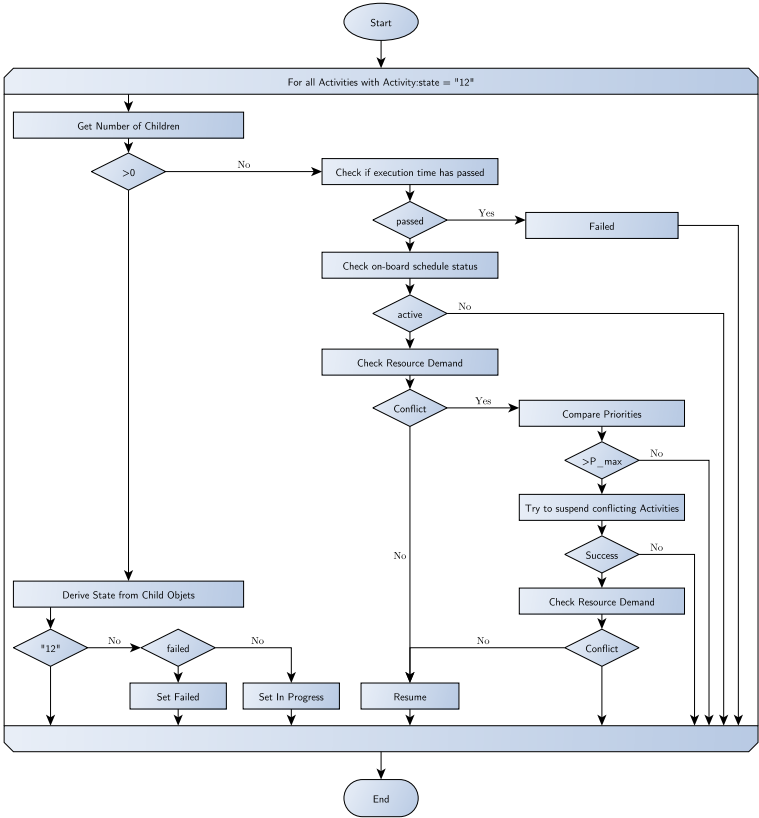


Figure B.12: Thread 12 State Machine

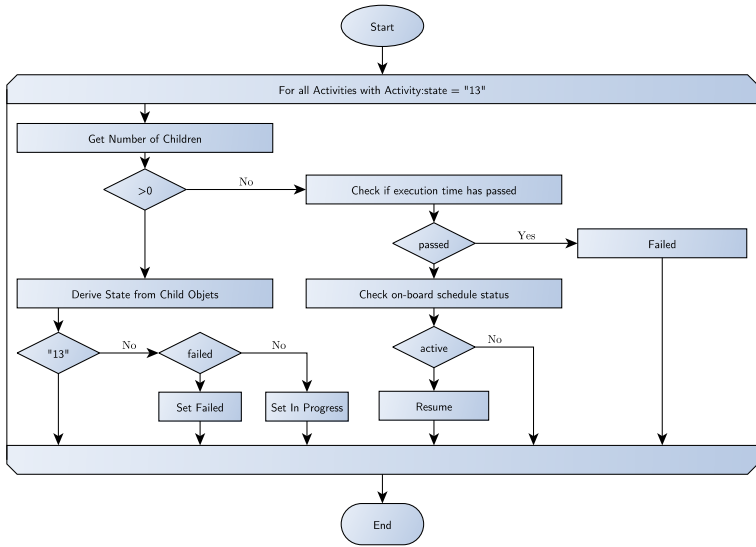


Figure B.13: Thread 13 State Machine

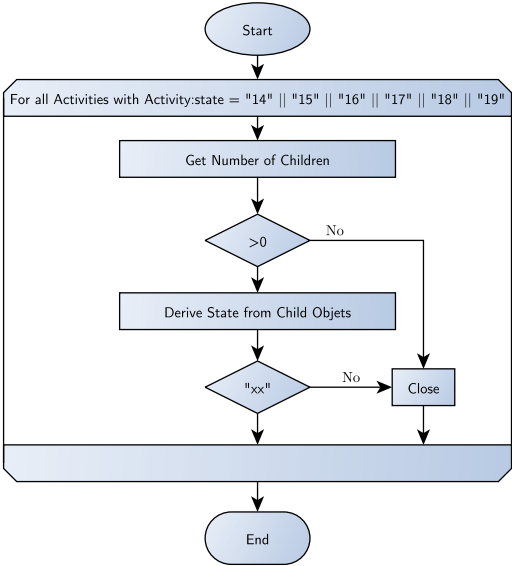


Figure B.14: Threads 14–19 State Machine

Space Debris

An aspect, that should be addressed by a work about satellite constellations, is the problem of space debris dissemination. Space around Earth is a very large but limited resource. And as always, humankind is exploiting that resource in a non-sustainable manner. Since the beginning of space flight in the late 1950th, the number of space debris objects is constantly growing.

In the year 1978, NASA scientist Donald J. Kessler described the growth of a fragment belt around earth as a result of colliding space objects in his article on *Collision frequency of artificial satellites* [79]. The scenario of cascading collisions of space objects that would result in a permanent fragment belt around earth is referred to as *Kessler syndrome* today. Such a belt would significantly affect space flight if not making it totally impossible for certain altitudes. Already back then Kessler warned of a “significant problem during the next century” [79].

Today, in 2021 space flight is still possible. Yet, it relies on a set of safety mechanisms, such as object tracking and active collision avoidance maneuvers. Furthermore, missions are supposed to be compliant with requirements for space debris mitigation such as ISO 24113 [45]. However, the numbers provided by *ESA’s Annual Space Environment Report* of 2019 are still alarming. Between the

beginning of tracking in 1960 and 2007 the number of known objects in orbit remained below 10,000. Since 2007 that number has more than doubled, mostly in low earth orbits ([51], p. 13, fig. 2.2(a)).

This increase in numbers could of course be due to improved detection mechanisms, but the major portion can surely be traced back to triggering events. As a matter of fact, the amount of known objects rises, because of the increasing number of satellites launched into orbit. Compared to 2016, the annual payload traffic into altitudes between 200 *km* and 1750 *km* has risen by roughly 100% ([51], p. 20, fig. 2.8). This goes along with a growing market and several companies such as SpaceX announcing the launch of large satellite fleets [62, 103, 63].

But not only satellite launches increase the amount of objects. More concerning are reckless operations or accidental events such as explosions or collisions of objects. Two prominent examples are a Chinese anti-satellite missile test in 2007 [30] and the collision of an Iridium satellite with a decommissioned Russian Kosmos-2251 satellite in 2009 [2]. Both events resulted in a noticeable rise of in-orbit fragments.

The second event further demonstrates what kind of risk a poor decommissioned or malfunctioning satellite could be for the safety of space flight. This is why mission failures especially in large altitudes are a complete taboo. To prevent that from happening space missions demand for a sense of responsibility, as well as they require reliable systems on space and on ground.

Bibliography

- [1] Dirk Abel. *Regelungstechnik*. Ed. by Aachener Forschungsgemeinschaft Regelungstechnik e.V. Aachen, Germany: Verlagsgruppe Mainz, 2009. ISBN: 3-8107-0067-3.
- [2] Joel Achenbach. *Debris From Satellites' Collision Said to Pose Small Risk to Space Station*. The Washington Post. Feb. 12, 2009. URL: <https://www.washingtonpost.com/wp-dyn/content/article/2009/02/11/AR2009021103387.html>.
- [3] Patricia Adam. *Agil in der ISO 9001*. Wiesbaden, Germany: Springer Gabler, 2020. ISBN: 978-3-658-28310-0. DOI: 10.1007/978-3-658-28311-7.
- [4] Eduardo Santana de Almeida. “Software Reuse and Product Line Engineering”. In: *Handbook of Software Engineering*. Ed. by Sungdeok Cha, Richard N. Taylor, and Kyochul Kang. Cham, Switzerland: Springer, 2019. ISBN: 978-3-030-00262-6.
- [5] Andreas Farley, Daniel Koch, Lukas Heiland, Marius Hauser, Max Hausch, Michael Erdenmann, Michael Voessner and Simon Hauser. “Student Project about the Development of a Mission Planning Tool for Satellite Operations”. University of Stuttgart, Germany, Oct. 2021.
- [6] Guillermo Arango. “A Brief Introduction to Domain Analysis”. In: *Proceedings of the 1994 ACM Symposium on Applied Computing - SAC '94*. Schlurnberger Laboratory for Com-

- puter Science. Phoenix, AZ, USA: ACM Press, 1994, pp. 42–46. DOI: 10.1145/326619.326656.
- [7] Niklas Arens. “Enhanced Development of a Disturbance Model for Small Satellite Simulation”. MA thesis. Stuttgart, Germany: University of Stuttgart Institute of Space Systems, Apr. 30, 2019.
- [8] Helmut Balzert. *Lehrbuch der Softwaretechnik. Entwurf, Implementierung, Installation und Betrieb*. 3rd ed. Heidelberg, Germany: Spektrum, 2011.
- [9] Bastian Bätz. “Design and Implementation of a Spacecraft Flight Software Framework”. PhD thesis. Stuttgart, Germany: University of Stuttgart Institute of Space Systems, Jan. 9, 2020.
- [10] Fabrice Bellard. *QEMU Wiki. Documentation/Platforms*. QEMU. Mar. 17, 2020. URL: <https://wiki.qemu.org/Documentation/Platforms>.
- [11] Mohamed Khalil Ben-Larbi et al. “Towards the Automated Operations of Large Distributed Satellite Systems. Part 1: Review and Paradigm Shifts”. In: *Advances in Space Research* (Aug. 2020). DOI: 10.1016/j.asr.2020.08.009.
- [12] Mohamed Khalil Ben-Larbi et al. “Towards the Automated Operations of Large Distributed Satellite Systems. Part 2: Classifications and Tools”. In: *Advances in Space Research* (Sept. 2020). DOI: 10.1016/j.asr.2020.08.018.
- [13] Barry W. Boehm. “Guidelines for Verifying and Validating Software Requirements and Design Specifications”. In: *Euro IFIP 79*. Redondo Beach, CA, USA, 1979, pp. 711–719.
- [14] André B. Bondi. “Characteristics of Scalability and Their Impact on Performance”. In: *Proceedings of the 2nd international workshop on Software and performance*. Ottawa, On-

- tario, Canada: Association for Computing Machinery, Sept. 2000, pp. 195–203. DOI: 10.1145/350391.350432.
- [15] Maximilian Böttcher et al. “Design and Implementation of a Wideband Back-Fire-Feed-System for an S-band Cassegrain Antenna”. In: *36th ESA Antenna Workshop on Antennas and RF Systems for Space Science*. Noordwijk, The Netherlands, Oct. 2015.
- [16] Maximilian Böttcher et al. “Design of a Low-Cost S/X Dual Band Stellite Ground Station for Small Satellite Missions”. In: *IAA/AAS Scitech 2020*. Moscow, Russia, Dec. 2020.
- [17] Alvise Braga-Illa. “Automatic satellite station-keeping”. In: *Journal of Spacecraft and Rockets* 6.4 (May 23, 2012), pp. 430–436. DOI: 10.2514/3.29674.
- [18] Nico Bucher. “Merging Spacecraft Software Development and System Tests: An Agile Verification Approach”. PhD thesis. Stuttgart, Germany: University of Stuttgart Institute of Space Systems, 2018.
- [19] Timothy A. Budd et al. “The design of a prototype mutation system for program testing”. In: *National Computer Conference*. American Federation of Information Processing Societies. Anaheim, CA, USA: AFIPS, June 1978, pp. 623–629.
- [20] Frank Buschmann et al. *Pattern-Oriented Software Architecture. A System of Patterns*. Vol. 1. Chichester, UK: Wiley, 2001. ISBN: 978-0471958697.
- [21] California Polytechnic State University. *CubeSat Design Specification*. Version 13. San Luis Obispo, CA, USA, Feb. 2, 2014.
- [22] Claude Cazenave, Francis Rodor, and Bjoern Kircher. *SIMTG Kernel User Manual*. SIMTG-UM-0008-ASTR. Version 2.2. Airbus DS. Mar. 9, 2015.

- [23] CCSDS. *Mission Operations Message Abstraction Layer*. CCSDS 521.0-B-2. Consultative Committee for Space Data Systems. Washington D.C., USA, Mar. 2013.
- [24] CCSDS. *Space Packet Protocol*. CCSDS 133.0-B-1. Consultative Committee for Space Data Systems. Washington D.C., USA, Sept. 1, 2003.
- [25] CCSDS. *XML Telemetric and Command Exchange (XTCE)*. CCSDS 660.0-B-1. Consultative Committee for Space Data Systems. Washington D.C., USA, Oct. 2007.
- [26] Scott Chacon and Ben Straub. *Pro Git*. 2nd ed. New York, NY, USA: Apress, 2014. ISBN: 978-1-4842-0076-6.
- [27] Thomas J. Cheatham and Lee Mellinger. “Testing Object-Oriented Software Systems”. In: *Proceedings of the 1990 ACM Annual Conference on Cooperation*. Vol. 18. Association for Computing Machinery. New York, NY, USA, Jan. 1990, pp. 161–165. DOI: 10.1145/100348.100373.
- [28] Yaofei Chen et al. “An Empirical Study of Programming Language Trends”. In: *IEEE Software* 22.3 (June 2005), pp. 72–79. DOI: 10.1109/MS.2005.55.
- [29] Colin Cherry. *On Human Communication*. Cambridge, MA, USA: The M.I.T. Press, 1966.
- [30] Craig Covault. *Chinese Test Anti-Satellite Weapon*. Aviation Week. Jan. 17, 2007. URL: https://web.archive.org/web/20070128075259/http://www.aviationweek.com/aw/generic/story_channel.jsp?channel=space%5C&id=news/CHI01177.xml.
- [31] Krzysztof Czarnecki. “Generative Programming. Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models”. PhD thesis. Ilmenau, Germany: Department of Computer Science

- and Automation, Technical University of Ilmenau, Aug. 9, 1999.
- [32] Wolfgang Dahmen and Arnold Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. 2nd ed. Berlin, Germany: Springer, 2008. ISBN: 978-3-540-76492-2.
- [33] Fabienne Delhaise et al. “Spacecraft and Payload Data Handling”. In: *ESA Special Publication SP-1295* (Nov. 2, 2007), pp. 1–13.
- [34] Jeremy Dick, Elizabeth Hull, and Ken Jackson. *Requirements Engineering*. 4th ed. Cham, Switzerland: Springer, 2017. ISBN: 978-3-319-61073-3.
- [35] Hans Dodel and René Wörfel. *Satellitenfrequenzkoordinierung*. Heidelberg, Germany: Springer, 2012. ISBN: 978-3-642-29202-6.
- [36] Ecma. *The JSON data interchange syntax*. ECMA-404. Ecma International. Geneva, Switzerland, Dec. 2017.
- [37] ECSS. *Glossary of Terms*. ECSS-S-ST-00-01C. European Cooperation for Space Standardization. Noordwijk, The Netherlands, Oct. 1, 2012.
- [38] ECSS. *Space engineering: Simulation Modelling Platform - Volume 1: Principles and Requirements*. ECSS-E-TM-40-07. European Cooperation for Space Standardization. Noordwijk, The Netherlands, Jan. 25, 2011.
- [39] ECSS. *Space engineering: SpaceWire – Links, nodes, router-sand networks*. ECSS-E-ST-50-12C. European Cooperation for Space Standardization. Noordwijk, The Netherlands, May 22, 2019.
- [40] ECSS. *Space engineering: SpaceWire – Remote memory access protocol*. ECSS-E-ST-50-52C. European Cooperation for Space Standardization. Noordwijk, The Netherlands, Feb. 5, 2010.

- [41] ECSS. *Space engineering: Telemetry and telecommand packet utilization*. ECSS-E-ST-70-41C. European Cooperation for Space Standardization. Noordwijk, The Netherlands, Apr. 15, 2016.
- [42] ECSS. *Space engineering: Telemetry and telecommand packet utilization*. ECSS-E-ST-70-41C. European Cooperation for Space Standardization. Noordwijk, The Netherlands, Apr. 15, 2016.
- [43] ECSS. *Space engineering: Testing*. ECSS-E-ST-10-03C. European Cooperation for Space Standardization. Noordwijk, The Netherlands, June 1, 2012.
- [44] ECSS. *Space product assurance: Software product assurance*. ECSS-Q-ST-80C Rev.1. European Cooperation for Space Standardization. Noordwijk, The Netherlands, Feb. 15, 2017.
- [45] ECSS. *Space sustainability*. ECSS-U-AS-10C. European Cooperation for Space Standardization. Noordwijk, The Netherlands, Dec. 3, 2019.
- [46] Manfred von Ehrenfried. *Apollo Mission Control*. Cham, Switzerland: Springer, 2018. ISBN: 978-3-319-76683-6. DOI: 10.1007/978-3-319-76684-3.
- [47] Jens Eickhoff. *Simulating Spacecraft Systems*. Heidelberg, Germany: Springer, 2009. ISBN: 978-3-642-01275-4.
- [48] Jens Eickhoff. *The FLP Microsatellite Platform. Flight Operations Manual*. Springer Aerospace Technology. Cham, Switzerland: Springer, 2016. ISBN: 978-3-319-23502-8. DOI: 10.1007/978-3-319-23503-5.
- [49] Jens Eickhoff, Harald Eisenmann, and Oliver Kienzler. "TINA - Knowledgebased Mission Planning for Future Spacecrafts and their Autonomous Operation". In: *WIT Transactions on Information and Communication Technologies* 19 (1997). DOI: 10.2495/AI970421.

- [50] Jens Eickhoff et al. “Constellations Research using simulated FLP-based Satellites”. In: *SpaceOps Conference*. Daejeon, Korea, May 20, 2016. DOI: 10.2514/6.2016-2544.
- [51] ESA. *ESA’s Annual Space Environment Report*. Tech. rep. GEN-DB-LOG-00271-OPS-SD. Version 3.2. Darmstadt, Germany: European Space Operations Centre (ESOC), July 17, 2019.
- [52] ESA. *SCOS-2000 Database Import ICD*. EGOS-MCS-S2K-ICD-0001. Version 6.9. European Space Agency. July 6, 2010.
- [53] Annette Froehlich. *Legal Aspects Around Satellite Constellations*. Ed. by European Space Policy Institute. Cham, Switzerland: Springer, 2019. ISBN: 978-3-030-06027-5. DOI: 10.1007/978-3-030-06028-2.
- [54] Daniel Galla et al. “The Educational Platform SOURCE - A CubeSat Mission on Demise Investigation Using In-Situ Heat Flux Measurements”. In: *70th International Astronautical Congress (IAC)*. IAC-19,E1,IP,24,x53779. International Astronautical Federation (IAF). Washington, D.C., USA, Oct. 2019.
- [55] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994. ISBN: 9780321700698.
- [56] Gonzalo Garcia. “A New Approach for Satellite Operations and Testing Automation using Python”. San Diego, CA, USA. In: *AIAA SPACE 2008 Conference*. AIAA 2008-7864. American Institute of Aeronautics and Astronautics. Sept. 2008. DOI: 10.2514/6.2008-7864.
- [57] Eberhard Gill. *Together in space: Potentials and challenges of distributed space systems*. Inaugural speech. Delft University of Technology. Sept. 17, 2008. URL: <http://resolver.tudelft.nl/uuid:13b5b1cf-1d3e-44b0-a724-44896cf157db>.

- [58] Jake Goulding. *What is Rust and why is it so popular?* Stack Overflow. Jan. 20, 2020. URL: <https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/>.
- [59] GSOC. *Planning Modelling Language*. Ed. by German Space Operations Center. German Areospace Center. July 2010. URL: https://www.dlr.de/rb/en/Portaldata/38/Resources/dokumente/GSOC_dokumente/RB-MIB/GSOC_Modelling_Language.pdf.
- [60] Noopur Gupta. *Embracing JUnit 5 with Eclipse*. Eclipse Foundation. 2017. URL: https://www.eclipse.org/community/eclipse_newsletter/2017/october/article5.php.
- [61] Klaus Henning, Arno Gramatke, and Julia Sabine Jakobs. *Informationsmanagement im Maschinenwesen. Foundations, Applications and Challenges*. 8th ed. Aachen, Germany: Druckerei und Verlagsgruppe Mainz, 2008. ISBN: 3-86073-735-X.
- [62] Caleb Henry. *OneWeb's first six satellites in orbit following Soyuz launch*. SpaceNews. Feb. 27, 2019. URL: <https://spacenews.com/first-six-oneweb-satellites-launch-on-soyuz-rocket/>.
- [63] Caleb Henry. *SpaceX launches 60 Starlink satellites, begins constellation buildout*. SpaceNews. May 23, 2019. URL: <https://spacenews.com/spacex-launches-60-starlink-satellites-begins-constellation-buildout/>.
- [64] Mark D. Hill. "What is Scalability". In: *ACM SIGARCH Computer Architecture News* 18 (4 Dec. 2, 1990), pp. 18–21. DOI: 10.1145/121973.121975.
- [65] William E. Hill. "My wife and my mother-in-law. They are both in this picture - find them". In: *Puck*. Vol. 78. Nov. 6, 1915, p. 11.

- [66] Gerald R. Hintz. *Orbital Mechanics and Astrodynamics*. Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-09443-4.
- [67] Joseph Howard, Dipak Oza, and Danford Smith. “Best Practices for Operations of Satellite Constellations”. In: *SpaceOps 2006 Conference*. AIAA 2006-5866. American Institute of Aeronautics and Astronautics. Rome, Italy: American Institute of Aeronautics and Astronautics, June 2006. DOI: 10.2514/6.2006-5866.
- [68] William E. Howden. “Completeness Criteria for Testing Elementary Program Functions”. In: *Proceedings of the 5th International Conference on Software Engineering*. IEEE, Mar. 1981, pp. 235–243.
- [69] IEEE. *Standard for Software and System Test Documentation*. IEEE Std 829-2008. The Institute of Electrical and Electronics Engineers. New York, NY, USA, July 18, 2008.
- [70] IEEE. *Standard for Software Test Documentation*. IEEE Std 829-1998. The Institute of Electrical and Electronics Engineers. New York, NY, USA, Sept. 16, 1998.
- [71] IEEE. *Systems and software engineering - Software life cycle processes*. Ed. by ISO. IEEE Std 12207:2008. The Institute of Electrical and Electronics Engineers. Geneva, Switzerland, Feb. 1, 2008.
- [72] IEEE. *The Authoritative Dictionary of IEEE Standards Terms*. The Institute of Electrical and Electronics Engineers. New York, NY, USA, Dec. 2000. ISBN: 0-7381-2601-2.
- [73] IETF. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. Internet Engineering Task Force. Dec. 2018. URL: <https://datatracker.ietf.org/doc/html/rfc8259>.

- [74] ITU. *Data Networks and Open System Communications: Open System Interconnection - Model and Notation*. X.200. ITU-T Recommendation. International Telecommunication Union. July 1994.
- [75] Stephen C. Johnson. “Lint, a C Program Checker”. In: *Computer Science Technical Report*. Vol. 65. Bell Laboratories. 1978.
- [76] Steven Johnson. *Emergence. The Connected Lives of Ants, Brains, Cities, and Software*. New York, USA: Scribner, 2001. ISBN: 978-0-684-86875-2.
- [77] Eric M. Jones. *Apollo 12 Lunar Surface Journal*. Apr. 27, 2013. URL: <https://www.hq.nasa.gov/alsj/a12/cuff12.html>.
- [78] Jonas Keim et al. “Commissioning of the Optical Communication Downlink System OSIRISv1 on the University Small-Satellite Flying Laptop”. In: *70th International Astronautical Congress*. Washington D.C., USA, Oct. 2019.
- [79] Donald J. Kessler and Burton G. Cour-Palais. “Collision frequency of artificial satellites: The creation of a debris belt”. In: *JGR Space Physics* 83.A6 (Feb. 22, 1978), pp. 2637–2646. DOI: 10.1029/JA083iA06p02637.
- [80] Alex Kirlik. “Modeling Strategic Behavior in Human-Automation Interaction. Why an “Aid” Can (and Should) Go Unused”. In: *Human Factors* 35 (2 June 1, 1993), pp. 221–242. DOI: 10.1177/001872089303500203.
- [81] Kai-Sören Klemich et al. “The Flying Laptop University Satellite Mission: Ground Infrastructure and Operations after one Year in Orbit”. In: *Deutscher Luft- und Raumfahrtkongress 2018*. Deutsche Gesellschaft für Luft- und Raumfahrt - Lilienthal-Oberth e.V. Friedrichshafen, Germany, Nov. 9, 2018. DOI: 10.25967/480190.

- [82] Gerhard Krieger et al. “TanDEM-X: A Satellite Formation for High-Resolution SAR Interferometry”. In: *IEEE Transactions on Geoscience and Remote Sensing* 45.11 (Nov. 2007), pp. 3317–3341.
- [83] Philippe B. Kruchten. “The 4+1 View Model of Architecture”. In: *IEEE Software* 12.6 (Nov. 1995), pp. 42–50. DOI: 10.1109/52.469759.
- [84] Jürg Kuster et al. *Handbuch Projektmanagement*. Berlin, Germany: Springer, 2019. ISBN: 978-3-662-57877-3. DOI: 10.1007/978-3-662-57878-0.
- [85] Jean-Claude Laprie. “Dependable Computing and Fault Tolerance. Concepts and Terminology”. In: *25th International Symposium on Fault-Tolerant Computing*. Pasadena, CA, USA: IEEE, June 1995. DOI: 10.1109/FTCSH.1995.532603.
- [86] Kai Leidig and Jens Eickhoff. “Microsatellite Simulation for Constellation Research”. In: *AIAA Modeling and Simulation Technologies Conference*. AIAA 2017-1553. American Institute of Aeronautics and Astronautics. Grapevine, TX, USA, Jan. 2017. DOI: 10.2514/6.2017-1553.
- [87] Kai Leidig et al. “Multi-Mission Operations System supporting Satellite Constellations”. In: *16th International Conference on Space Operation*. SpaceOps-2020,4,14,x281. International Astronautical Federation (IAF). Cape Town, South Africa, May 2021.
- [88] Panagiotis Louridas. “Static Code Analysis”. In: *IEEE Software* 23.4 (July 17, 2006), pp. 58–61. DOI: 10.1109/MS.2006.114.
- [89] Dewi Mairiza, Didar Zowghi, and Nurie Nurmuliani. “An Investigation into the Notion of Non-Functional Requirements”. In: *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*. Sierre, Switzerland, Jan. 2010. DOI: 10.1145/1774088.1774153.

- [90] Edith Maurer et al. “TerraSAR-X Mission Planning System. Automated Command Generation for Spacecraft Operations”. In: *IEEE Transactions on Geoscience and Remote Sensing* 48 (Feb. 2010), pp. 642–648. DOI: 10.1109/TGRS.2010.2040699.
- [91] Ernst Messerschmid and Stefanos Fasoulas. *Raumfahrtsysteme*. Berlin, Germany: Springer, 2017. ISBN: 978-3-662-49637-4.
- [92] Sreeja Nag et al. “Autonomous Scheduling of Agile Spacecraft Constellations with Delay Tolerant Networking for Reactive Imaging”. In: *International Conference on Automated Planning and Scheduling SPARK Workshop* (July 2019).
- [93] Kshirasagar Naik and Priyadarshi Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. Wiley, 2008. ISBN: 9780471789116.
- [94] James M. Neighbors. “Software Construction Using Components”. PhD thesis. Irvine, CA, USA: Department of Information and Computer Science, University of California, 1980.
- [95] Gregorius Ongo and Gede Putra Kusuma. “Hybrid Database System of MySQL and MongoDB in Web Application Development”. In: *International Conference on Information Management and Technology (ICIMTech)*. Jakarta, Indonesia: IEEE, Sept. 5, 2018. DOI: 10.1109/ICIMTech.2018.8528120.
- [96] Rafael Vázquez Osorio et al. “SCOS-2000 Release 4.0. Multi-Mission/Multi-Domain Capabilities in ESA SCOS-2000 MCS Kernel”. In: *2006 IEEE Aerospace Conference*. Big Sky, MT, USA: IEEE, July 24, 2006. DOI: 10.1109/AERO.2006.1656141.
- [97] Sebastian Oster. “Feature Model-based Software Product Line Testing”. PhD thesis. Darmstadt, Germany: TU Darmstadt, Dec. 16, 2011.

- [98] Thorsten Ottosen. *Pointer Container Library*. Ed. by boost C++ Libraries. 2007. URL: https://www.boost.org/doc/libs/1_74_0/libs/ptr_container/doc/ptr_container.html.
- [99] Raja Parasuraman, Thomas B. Sheridan, and Christopher D. Wickens. “A Model for Types and Levels of Human Interaction with Automation. Part A: Systems and Humans”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 30 (3 May 2000), pp. 286–297. DOI: 10.1109/3468.844354.
- [100] Steve Pearson, Simon Reid, and Wernke zur Borg. “A Full End-to-end Automation Chain with MOIS, PLUTO, MATIS, SMF and SCOS-2000”. In: *SpaceOps 2014 Conference*. Pasadena, CA, USA, May 2, 2014. DOI: 10.2514/6.2014-1833.
- [101] Chris Peat. *Flying Laptop - Orbit*. Heavens-Above GmbH. Oct. 7, 2021. URL: <https://heavens-above.com/orbit.aspx?satid=42831&lat=0&lng=0&loc=Unspecified&alt=0&tz=UCT>.
- [102] Deqayne E. Perry and Gail E. Kaiser. “Adequate Testing and Object-Oriented Programming”. In: *Journal of Object-Oriented Programming* 2.5 (1990).
- [103] Kattia Flores Pozo and Adriana Fukuzato. “Too Many Satellites to Operate? How Planet Successfully Operates 100’s of Satellites using Agile Aerospace”. In: *69th International Astronautical Congress (IAC)*. IAC-18.B6.2.1X46924. International Astronautical Federation (IAF). Bremen, Germany, Oct. 2018.
- [104] Robert Ramey. *Serialization. Overview*. Ed. by boost C++ Libraries. 2004. URL: https://www.boost.org/doc/libs/1_72_0/libs/serialization/doc/index.html.
- [105] Brandon Craig Rhodes. *PyEphem. Astronomy Library for Python*. 2020. URL: <https://rhodesmill.org/pyephem/>.

- [106] Brandon Craig Rhodes. *Skyfield. Elegant Astronomy for Python*. 2020. URL: <https://rhodesmill.org/skyfield/>.
- [107] Shamim Ripon et al. “Automated Requirements Extraction and Product Configuration Verification for Software Product Line”. In: *Automated Software Testing*. Ed. by Ajay Kumar Jena, Himansu Das Durga, and Prasad Mohapatra. Singapore: Springer, 2020. ISBN: 978-981-15-2455-4.
- [108] Stefan Röß. “Development of a Simple Env/Dyn-Model for the Simulation of a Small Satellite”. MA thesis. Stuttgart, Germany: University of Stuttgart Institute of Space Systems, Nov. 9, 2016.
- [109] RTEMS Project and contributors. *RTEMS User Manual (5.1)*. 2020. URL: <https://ftp.rtems.org/pub/rtems/releases/5/5.1/docs/html/user/index.html>.
- [110] Jeff Rubin and Dana Chisnell. *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*. 2nd ed. Indianapolis, IN, USA: Wiley, 2008. ISBN: 978-0-470-18548-3.
- [111] Shadman Salih. “Selection of Computer Programming Languages for Developing Distributed Systems”. PhD thesis. Leicester, UK: Department of Software Engineering, University of De Montfort, May 15, 2014.
- [112] Luca Save. “Not all or nothing, not all the same: classifying automation in practice. Safety and Automation”. In: *Hind-Sight* 20 (Dec. 2014). Ed. by Tzvetomir Blajev, pp. 64–68.
- [113] Luca Save and Beatrice Feuerberg. “Designing Human-Automation Interaction. A new level of Automation Taxonomy”. In: *Human Factors: a view from an integrative perspective*. Ed. by Dick de Waard. 2012.
- [114] Benjamin Schoch et al. “Towards a CubeSat Mission for a Wideband Data Transmission in E-Band”. In: *IEEE Space*

- Hardware and Radio Conference (SHaRC)*. IEEE. San Antonio, TX, USA, Jan. 29, 2020.
- [115] Ken Schwaber and Jeff Sutherland. *The Scrum Guide*. Nov. 2017. URL: <https://scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf>.
- [116] SFC. *git homepage*. Ed. by Software Freedom Conservancy. 2020. URL: <https://git-scm.com/>.
- [117] Graeme B. Shaw, D. W. Miller, and D. E. Hastings. “Generalized Characteristics of Communication, Sensing, and Navigation Satellite Systems”. In: *Journal of Spacecraft and Rockets* 37.6 (Dec. 2000), pp. 801–811. DOI: 10.2514/2.3638.
- [118] Thomas B. Sheridan and William L. Verplank. *Human and Computer Control of Undersea Teleoperators*. Tech. rep. Cambridge, MA, USA: Man-Machine Systems Laboratory, MIT, July 1978.
- [119] Michael Shpilt. *9 Must Decisions in Web Application Development*. Sept. 2019. URL: <https://michaelscodingspot.com/web-application-development/#spa>.
- [120] Marc Costa Sitjà. “SPICE for ESA Planetary Missions. geometry and visualization support to studies, operations and data analysis within your reach”. In: *SpaceOps 2018 Conference*. Marseille, France: AIAA, May 25, 2018. DOI: 10.2514/6.2018-2553.
- [121] Harry M. Sneed and Mario Winter. *Testen objektorientierter Software. Das Praxishandbuch für den Test objektorientierter Client-Server-Systeme*. München, Germany: Hanser, 2002. ISBN: 3-446-21820-3.
- [122] Stephan Speidel. “Conceptual Approach of a Thermal Model for Satellite Simulation”. MA thesis. Stuttgart, Germany: University of Stuttgart Institute of Space Systems, Apr. 15, 2018.

- [123] Herbert Stachowiak. *Allgemeine Modelltheorie*. Vienna, Austria: Springer, 1973. ISBN: 3-211-81106-0.
- [124] Bjarne Stroustrup. *The C++ Programming Language*. 3rd ed. Boston, MA, USA: Addison-Wesley, 2001. ISBN: 0-201-88954-4.
- [125] Terma. *Data sheet - CCS5 and TSC*. Aarhus, Denmark, 2020.
- [126] Georg Erwin Thaller. *Software-Test*. Hannover, Germany: Heinz Heise, 2000. ISBN: 3-88229-183-4.
- [127] The PostgreSQL Global Development Group. *PostgreSQL Documentation*. Aug. 12, 2021. URL: <https://www.postgresql.org/docs/>.
- [128] Thomas Uhlig, Florian Sellmaier, and Michael Schmidhuber. *Spacecraft Operations*. Ed. by German Space Operations Center (GSOC). Ed. by German Aerospace Center (DLR). Vienna, Austria: Springer, 2015. ISBN: 978-3-7091-1802-3. DOI: 10.1007/978-3-7091-1803-0.
- [129] Yuri Ulybyshev. “Long-Term Formation Keeping of Satellite Constellation Using Linear-Quadratic Controller”. In: *Journal of Guidance, Control, and Dynamics* 21.1 (May 23, 2012), pp. 109–115. DOI: 10.2514/2.4204.
- [130] David A. Vallado and Paul Crawford. “SGP4 Orbit Determination”. In: *AIAA/AAS Astrodynamics Specialist Conference*. Honolulu, Hawaii, USA: AIAA, June 15, 2012. DOI: 10.2514/6.2008-6770.
- [131] Anthony Walsh et al. “The European Ground Systems Common Core (EGS-CC) Initiative”. In: *SpaceOps 2012 Conference*. Stockholm, Sweden, June 2012. DOI: 10.2514/6.2012-1282732.
- [132] Joachim Weiß. *Duden - Das Neue Lexikon*. 3rd ed. Vol. 5: Indi - Lau. Mannheim, Germany: Bibliographisches Institut & F. A. Brockhaus AG, 1996. ISBN: 3-411-04353-9.

- [133] Sebastian Wenzel et al. “Pointing Enhancement for an Optical Laser Downlink Using Automated Image Processing”. In: *Small Satellite Conference*. SSC20-WKVIII-05. Logan, UT, USA, July 28, 2020.
- [134] James R. Wertz. *Space Mission Engineering: The New SMAD*. Ed. by James R. Wertz. Ed. by David F. Everett. Ed. by Jeffery J. Puschell. Hawthorne, CA, USA: Microcosm Press, 2011. ISBN: 978-1-881-883-15-9.
- [135] Karl Wieggers and Joy Beatty. *Software Requirements*. 3rd ed. Redmond, WA, USA: Microsoft Press, 2013.
- [136] Wikipedia. *Complexity*. Dec. 30, 2019. URL: <https://en.wikipedia.org/wiki/Complexity>.
- [137] Wikipedia. *JavaScript Object Notation*. Oct. 3, 2021. URL: https://de.wikipedia.org/wiki/JavaScript_Object_Notation.
- [138] Lukas Wunderlich. “Systemsimulation einer Satellitenkonstellation basierend auf dem Kleinsatelliten Flying Laptop”. BA thesis. Stuttgart, Germany: University of Stuttgart Institute of Space Systems, June 2020.
- [139] Wu Yu. “Modification and Installation of a Small Satellite On-Board Software on an Emulated Processing Unit”. MA thesis. Stuttgart, Germany: University of Stuttgart, Institute of Space Systems, Apr. 17, 2017.
- [140] Olivier Zanon. “The SimTG Simulation Modeling Framework: A domain specific language for space simulation”. In: *2011 Spring Simulation Multi-conference*. DBLP. Boston, MA, USA, Jan. 2011.

Image References

Figure (4.1): William E. Hill. *My wife and my mother-in-law*. In: *Puck*. Vol. 78. Nov. 6, 1915, p. 11, GRANGER - Historical Picture Archive / Alamy Stock Photo, book print and/or digital license.

Figures (5.8), (5.12), (5.17), (5.34), (6.2), (6.16), and (6.27) use icons from *Flaticon.com*.

Index

- A**
- Abstraction 155
 - Ground-to-Ground 159
 - Space-to-Ground.. 145, 159, 179, 181
 - Abstraction Layer 155
 - Acceptance Test 75
 - Activity 130, 139, 177, 181
 - Interface 159
 - Link 208
 - Maneuver 210
 - State 198
 - Agent 98, 130
 - Architecture 48, 142
 - Automation 91, 130
 - Taxonomy 96
- B**
- Backlog 46
- C**
- Class 73
 - Test 73
 - Command 156
 - Interface 157
 - Complexity 3
 - Consistency 107
 - Constellation 2, 26
 - Planning Tool 228
- D**
- Data Handling Simulation .. 253
 - Debris 323
 - Distributed System 25
 - Domain 50
 - Analysis 52, 112
 - Design 56
 - Engineering 50
- E**
- Emulation 246
 - End-to-End Test 246
 - Entity 163
 - Environment Dynamics 240
 - Epic 48
 - Error 64
- F**
- Failure 64

Fault 64
 Feature 53
 Model 53
 Tree 53, 173
 Flight Dynamics 126
 Flying Laptop 12
 Formation 1

G

Graphical User Interface 225
 Ground Segment 114

I

Implementation 57
 Integration Test 74
 Interface 153

L

Link 208

M

Maneuver 210
 Middleware 150
 Mission
 Control 116
 Information Base .. 109, 175
 Planning 133, 179
 Planning Tool 212
 Model 82
 MoSCoW 48

O

On-Board Computer Model . 249

On-Board Schedule 205
 Operations 36
 Orchestration 173, 231

P

Pass 208
 Project Organization 41

R

Reliability 106
 Requirement 55, 90
 Resource 188

S

Scalability 100
 Schedule 134, 163, 223
 Scrum 43
 Serialization 112, 154, 170
 SimTG 76
 Simulated Mission Time 86
 Simulation 76
 Model 238
 Runtime 85
 Simulator 256
 Session Time 85
 Space
 -to-Grnd. Abstraction . 145,
 159, 179, 181
 Segment 113
 System 112, 173
 System 115
 Information Base 173

Simulation	238
Test	75

T

Telemetry Archive	123
Test	
Environment	71
Item	71
Testing	63

U

Unit Test	73
-----------------	----

V

V-Model	69
Validation	64
Verification	64
Version Control	60
Virtual Machine	247

Z

Zulu-Time	85
-----------------	----

Each space system is divided into two main parts: the space segment, and the ground segment. While most space segments used to consist of just one single satellite, this thesis considers the growing case of a space segment featuring an arbitrary multiple of satellites. This is the case under two circumstances: The space segment is either a constellation, or a number of independent satellite missions are operated in parallel, which is referred to as multi-mission operations. From an operational perspective this leads to a series of challenges, which are addressed, handled, and solved by the work described in this thesis.



Following a holistic approach, this work addresses the design of a system for the operations of multiple satellites in parallel. The work was done under the assumption that a complex system like a constellation cannot be operated efficiently without automation, where automation is not just understood as an additional feature, but as a holistic process that involves the entire space system including the complete ground infrastructure, as well as the operated satellites.

