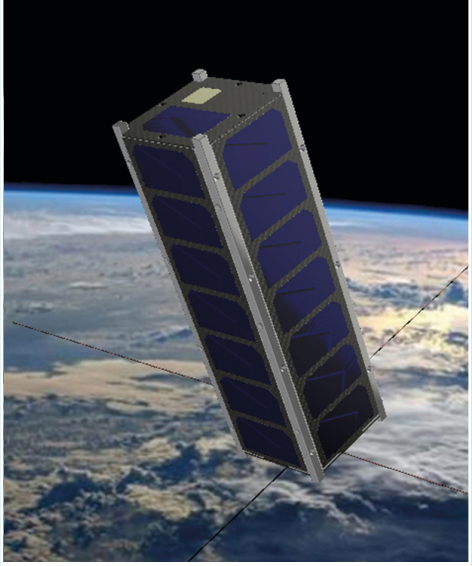
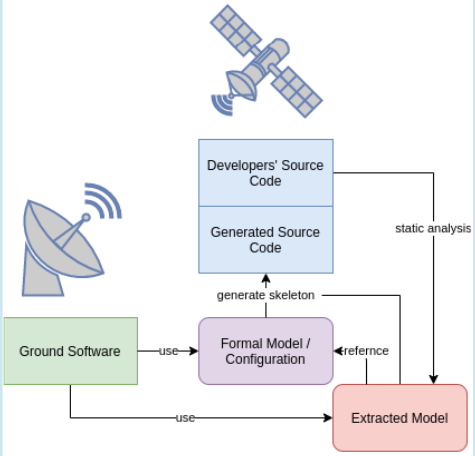


Institut für Informatik
Lehrstuhl für Informationstechnik
für Luft- und Raumfahrt
Prof. Dr. Sergio Montenegro



Research in Aerospace Information Technology

Julius-Maximilians-
UNIVERSITÄT WÜRZBURG

RAIT 2

Frank Flederer

CORFU
An Extended Model-Driven Framework for Small Satellite Software with Code Feedback

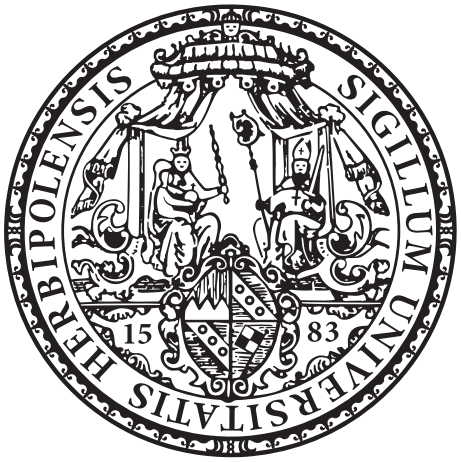
CORFU — An Extended Model-Driven Framework for Small Satellite Software with Code Feedback

---

Frank Flederer

October 18th, 2021

Julius-Maximilians-Universität Würzburg



Fakultät für Mathematik und Informatik
Informatik VIII: Aerospace Information Technology

Dissertation

CORFU — An Extended Model-Driven Framework for Small Satellite Software with Code Feedback

Frank Flederer

1. Reviewer Prof. Dr-Ing. Sergio Montenegro
Julius-Maximilians-Universität Würzburg

2. Reviewer Prof. Dr. Reiner Kolla
Julius-Maximilians-Universität Würzburg

Supervisor Prof. Dr-Ing. Sergio Montenegro

October 18th, 2021

Frank Flederer

CORFU — *An Extended Model-Driven Framework for Small Satellite Software with Code Feedback*
Dissertation, October 18th, 2021
Reviewers: Prof. Dr-Ing. Sergio Montenegro and Prof. Dr. Reiner Kolla
Supervisor: Prof. Dr-Ing. Sergio Montenegro

Julius-Maximilians-Universität Würzburg
Informatik VIII: Aerospace Information Technology
Institut für Informatik
Fakultät für Mathematik und Informatik
Emil-Fischer-Straße 70
97074 Würzburg

Abstract

NOTE: For better readability, I use *we* instead of *I* in this document.

Corfu is a framework for satellite software, not only for the onboard part but also for the ground. Developing software with Corfu follows an iterative model-driven approach. The basis of the process is an engineering model. Engineers formally describe the basic structure of the onboard software in configuration files, which build the engineering model. In the first step, Corfu verifies the model at different levels. Not only syntactically and semantically but also on a higher level such as the scheduling.

Based on the model, Corfu generates a software scaffold, which follows an application-centric approach. Software images onboard consist of a list of applications connected through communication channels called topics. Corfu's generic and generated code covers this fundamental communication, telecommand, and telemetry handling. All users have to do is inheriting from a generated class and implement the behavior in overridden methods. For each application, the generator creates an abstract class with pure virtual methods. Those methods are callback functions, e.g., for handling telecommands or executing code in threads.

However, from the model, one can not foresee the software implementation by users. Therefore, as an innovation compared to other frameworks, Corfu introduces feedback from the user code back to the model. In this way, we extend the engineering model with information about functions/methods, their invocations, their stack usage, and information about events and telemetry emission. Indeed, it would be possible to add further information extraction for additional use cases. We extract the information in two ways: assembly and source code analysis. The assembly analysis collects information about the stack usage of functions and methods.

On the one side, Corfu uses the gathered information to accomplished additional verification steps, e.g., checking if stack usages exceed stack sizes of threads. On the other side, we use the gathered information to improve the performance of onboard software. In a use case, we show how the compiled binary and bandwidth towards the ground is reducible by exploiting source code information at run-time.

Zusammenfassung

Corfu ist ein Framework für Satelliten-Software für beide Seiten: Space und Boden. Mit Corfu folgt die Softwareentwicklung einem iterativen modellgetriebenen Ansatz. Grundlage der Software-Entwicklung ist ein technisches Modell, das formell die grundlegende Struktur der Onboard-Software beschreibt. EntwicklerInnen beschreiben dieses Modell in Konfigurationsdateien, die von Corfu in verschiedenen Aspekten automatisch verifiziert werden, z.B. im Bereich des Scheduling.

Anhand des definierten Modells erstellt Corfu ein Quellcode-Gerüst. Die Onboard-Software ist in einzelne Applikationen aufgeteilt, die durch Kommunikationskanäle miteinander kommunizieren (Topics genannt). Generischer Code und der generierte Code implementieren bereits die Behandlung und Verwaltung der Topic-Kommunikation, Telekommandos, Telemetrie und Threads. Der generierte Code definiert pur-virtuelle Callback-Methoden, die BenutzerInnen in erbenden Klassen implementieren.

Das vordefinierte Modell kann allerdings nicht alle Implementierungsdetails der BenutzerInnen enthalten. Daher führt Corfu als Neuerung ein Code-Feedback ein. Hierbei werden anhand von statischer Analyse Informationen aus dem BenutzerInnen-Quellcode extrahiert und in einem zusätzlichen Modell gespeichert. Dieses extrahierte Modell enthält u.a. Informationen zu Funktionsaufrufen, Anomalien, Events und Stackspeicherverbrauch von Funktionen. Corfu extrahiert diese Informationen durch Quellcode- und Assembler-Analyse. Das extrahierte Modell erweitert das vordefinierte Modell, da es Elemente aus dem vordefinierten Modell referenziert.

Auf der einen Seite nutzt Corfu die gesammelten Informationen, um weitere Verifikationsschritte durchführen zu können, z.B. Überprüfen der Stack-Größen von Threads. Auf der anderen Seite kann die Nutzung von Quellcode-Informationen auch die Leistung verbessern. In einem Anwendungsfall zeigen wir, wie die Größe des kompilierten Programms sowie die genutzte Bandbreite für die Übertragung von Log-Event-Nachrichten durch das erweiterte Modell verringert werden kann.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ABI	Application Binary Interface
AOCS	Attitude and Orbit Control System
API	Application Programming Interface
AST	Abstract Syntax Tree
AUTOSAR	**Aut**omotive **O**pen **S**ystem **Ar**chitecture
BSP	Board Support Package
CCSDS	Consultative Committee for Space Data Systems
CI	Continuous Integration
CNES	Centre National d'Études Spatiales (National Centre for Space Studies)
CORFU	**Co**nfigurable Softwa**r**e **F**ramework for yo**u**
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DSR	Design Science Research
COTS	Components Off-The-Shelf
CPU	Central Processing Unit
DRE	Distributed, Real-time, and Embedded
DSL	Domain-Specific Language
DSML	Domain-Specific Modeling Language
ECSS	European Cooperation for Space Standardization
ESA	European Space Agency
FDIR	Fault Detection, Isolation, and Recovery
FIFO	First In First Out
GUI	Graphical User Interface
IDE	Integrated Development Engine
IPC	Inter Process Communication
JPL	Jet Propulsion Laboratory
JSF	Joint Strike Fighter (Air Vehicle C++ Coding Standards)
KISS	Keep It Short and Simple (or: Keep It Simple, Stupid)
LCM	Least Common Multiple
LEO	Low Earth Orbit
LEOP	Launch and Early Orbit Phase
LLOC	Logical Lines Of Code
MBD	Model-Based Development

MDD	Model-Driven Development
MISRA	Motor Industry Software Reliability Association
MIT	Massachusetts Institute of Technology
MMU	Memory Management Unit
NASA	National Aeronautics and Space Administration
OBSW	On-Board Software
OS	Operating System
OSI	Open Systems Interconnection
POSIX	Portable Operating System Interface
PROMELA	**Pro**cess **Me**ta **La**nguage
PUS	Packet Utilization Standard
RODOS	Real-time On-board Dependable Operating System
RTE	Round-Trip Engineering
RTEMS	Real-Time Executive for Multiprocessor Systems
SOFA	**Sof**tware **A**ppliances
SRI	Inertial Reference System
TC	Telecommand
TDD	Test-Driven Development
TM	Telemetry
TM/TC	Telemetry and Telecommands
UML	Unified Modeling Language
XSD	XML Schema Definition
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations
YAML	YAML Ain't Markup Language

to my family

Part I

Introduction

1 Introduction

This chapter gives a short introduction of the motivation for creating the software framework Corfu. It highlights the contribution and lists all the journal and conference publications of this work.

1.1 Motivation

Model-driven software frameworks improve software quality and shorten the development time. In the field of satellite software, there are already some approaches that go in the direction of model-driven development (MDD). However, there are limitations in current (model-driven) approaches, which we try to address in our framework Corfu.

1.1.1 Vision

Our vision is to have a unified software structure for satellite software, which takes all the burden of writing boilerplate code off users. Users describe the desired software structure and behavior concisely and abstractly: the (engineering) model. The framework generates most of the boilerplate code so that users can concentrate on implementing the mission-specific code.

By introducing information feedback from the source code into the model, we gain details about user implementations, which might influence the model, e.g., events, telemetry emission, or stack usage. Extracting this implementation information automatically and using it in the model can make the software more robust and performant.

1.1.2 Software Frameworks Support Reusability

The implementations of satellite software missions differ. They differ because of the type of mission, the used hardware, and the time epoch in which they are

developed. However, all implementations of satellite software are subject to some same conditions and requirements — independent of the specific mission and payload. For example, onboard software in satellites always has to collect data from its subsystems and periodically send them down to the ground to overview the overall vital state. In addition, they often have to provide a certain degree of autonomy because active ground contact is not available at any time. (Refer to B.1 for a list of general requirements of satellite software.)

Therefore, there is significant potential for reusing software components of satellite software. Even if the subsystems are different for distinct satellites, developers can pertain to the basic principle and the rough structure of onboard software. However, there must be a well-defined interface between the components, e.g., between the housekeeper, which collects the standard telemetry, and other subsystems. Software frameworks usually provide ways to define communication interfaces formally.

Software frameworks already come with a predefined rough structure and straightforward interface at which the user-defined code docks. However, for some satellite projects, simple frameworks might be too rigid. Frameworks should be flexible enough so that users can tailor them to the specific needs of space missions.

1.1.3 Model-Driven Development

The field of safety-critical software applies model-driven development more and more. It is also visibly arriving in the development of satellite software[5, 78].

Model-Driven Development vs. Model-Based Development The literature distinguishes between model-based (MBD) and model-driven development approaches (MDD). Both approaches require engineers to define software models in advance, i.e., before developers write code. Nyßen[71] differentiates model-based and model-driven engineering in his Ph.D. thesis as follows.

Model-driven software engineering uses tools (e.g., generators) for automatically processing the model to create software artifacts.

Model-based software engineering also creates software artifacts based on the model. In contrast to model-driven engineering, it is not necessary to achieve this with automatic tools. Instead, developers could transform the model into software by manually writing code.

Model-driven development is an approach that, on the one hand, gives developers freedom about configuring the framework and the final satellite software. On the other hand, it enables engineers to define the structure of the software before beginning with coding. Model-driven Development enforces engineers to early formalize the software structure, including the required components and the communication flow. Instead of defining models just for documentation, model-based frameworks can generate artifacts directly from the models, e.g., source code or documentation files. In addition, tools can use the information of the software's structure to accomplish early verification of the design.

For a detailed look into model-driven development, please have a look at chapter 3.

Problems of Classical Model-Driven Development As it is applied so far for satellite software, the classical model-driven development only knows one direction of information flow; from the model to the source code and documentation. However, models are just an abstraction of the final software, which does not contain all the information for generating the entire software. If we generate code from the model, it still needs to be refined manually by developers. In theory, developers extend the original model with additional information, which is the source code.

On the other side, it might be helpful to include source code information into the model. Using both types of information allows generators to access more accurate information about the software system and makes implementation details available to companion software like the ground software. As a consequence, the framework can improve the usage of resources, e.g., by determining the used stack size (see section 13.3) or by omitting constant strings from the software image and on the transmission path (see section 8.4.5). In addition, it allows extending the verification not only on the model but also to the user-written source code.

1.1.4 Agile Software Development

Usually, the development process for satellites still follows a traditional way. The classical (non-agile) way follows one direction, from the requirements and specifications to implementation, testing, and verification. However, there are some reports that agile software development is also making its way into satellite development[14, 98]. For example, Lill et al.[55, 56] have successfully applied agile development for satellite software. Even the ECSS recently released a handbook for agile software development in space engineering[31].

Agile software development bases on the "Manifesto for Agile Software Development," which has been established by several renowned software developers and engineers[6]. The manifest describes four values:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

ECSS' handbook for agile software development in space engineering[31] states:

> Agile is an iterative, time-boxed approach to software development in which a software product is built through an evolutionary process characterised by early and frequent deliveries of product increments, intensive customer collaboration and adaptive planning and goals, with the aim of responding to change in a rapid and flexible manner.

The central goal of agile software development is to have a working version of the software available at any time. Even if it is not finished — especially at the beginning of the development — early versions can already be tested and evaluated. Du to the non-strict plan, the development teams can quickly react to (requirement) changes or problems in the development process. Agile software development comes with additional requirements on model-driven frameworks, e.g., they also have to support an iterative way. That means the model is usually not fully defined when the software development begins. Instead, it evolves iteratively according to the agile process. Therefore, the software framework must ensure that modifying the model works seamlessly and that model changes do not impair the user code. Corfu achieves this by keeping the generated code and the user-written code in separate files. The user does not modify generated code and vice versa. Both parts, the generated and the user code, are only semantically connected via class inheritance.

Sweeting identifies in [93] several aspects of teams developing small satellites and their projects:

1. highly innovative staff
2. small, motivated teams
3. devolved responsibility, rigor, and quality
4. good team communications, close proximity

5. well-defined mission objectives and constraints
6. knowledgeable use of modern components
7. layered, failure-resilient system architecture
8. subsystem burn-in rather than component screening
9. short timescale (to prevent possible escalation of objectives)
10. design to cost
11. run by well-informed and responsive management personell

1.2 Approach of Corfu

This dissertation comprehensively describes the software framework Corfu. Corfu is a framework for developing satellite software in a model-driven way. It comes with a configuration format covering characteristics of small satellite software. The configuration files rely on the YAML syntax, which comprises nested elements of associations (key/value pairs), scalar data, and lists[79]. On top of this, we define a semantic schema in which developers describe satellite software.

Corfu organizes onboard software in separate reusable components, which we call applications. Each application implements a distinct functionality, which software images can incorporate. Applications consist of various elements; they can define threads, telecommands, telemetry, and so on, with their properties. Applications communicate via the pub/sub-communication middleware of Rodos. Therefore, applications define which topic types they subscribe to and to which they publish.

Nodes aggregate all the applications that should run on a processor. In modern satellites, several computers on board have to collaborate in order to accomplish the mission. Therefore, developers can define multiple nodes in the model, which combine different applications. Some applications might be part of every node, e.g., a housekeeping data collector, but some applications might be part only of particular nodes, for example, applications for payload.

Corfu uses information from the model in two different ways: First, it verifies the model, including a semantic check and more elaborated verification steps, such as scheduling analysis for threads in the system and analysis of (stack) memory usage. Second, it uses information from the model to generate parts of the source code. For example, the generated code contains the general structure of applications,

including the local distribution of telecommands and structs of telecommand and telemetry messages. The generated code leaves pure virtual methods, which have to be implemented by developers with the application's intended behavior. The set of pure virtual methods includes reactions on topic messages, telecommands, and code that is executed (periodically) in the context of threads.

Apart from the applications, the code generator of Corfu also creates code of nodes, which represent software images on processors. Such node code instantiates all configured applications and topics and configures them. The generated node code also includes the wiring of the applications through the topics. The model configuration already contains all the required information for nodes; it is unnecessary to contribute manual code.

Based on the user source code, which users contribute to the project, Corfu performs feedback back to the model. By applying static analysis on the source code, Corfu extracts information from the user implementation. Among other information, it includes stack requirements of functions, static values such as string literals, and behavioral aspects such as information about which reacting methods emit telemetry data.

Corfu gathers the extracted information in a model, which we call *extracted model*. The extracted model references elements from the engineering model, such as applications, threads, or telecommands. Therefore, the extracted model extends the engineering model by information from the source code. We call the combination of both the formal and the extracted model *extended model*.

In several use cases, we demonstrate how satellites can benefit from exploiting information from the extended model. By implementing a logging system, we show that we can improve the performance of the software. Here, the analyzer saves the message strings from the logging system into the extracted model. In this way, the message strings are also available at run-time, e.g., to the ground software. Thus, it is not necessary to store the entire string into the executable and transmit it. Instead, it is sufficient to store and transmit an identifier for the string.

By checking the user source code for data emission, like telemetry or topic publication, Corfu can verify whether the user used the correct reaction types. It does this by comparing the expected response types with the user source code. However, it does not check whether the user implemented everything right because the model does not contain complete information about what applications do.

By using information from the source code, Corfu can verify software parameters. One example is the stack size of threads. Corfu's analyzer determines the stack usage

of the user methods and functions and calculates the expected stack usage. Based on the result, Corfu can verify the stack sizes of threads, which engineers define at compile time.

Satellite software is not limited to the part on board. On the ground side, satellite missions need to implement corresponding ground software, which can also exploit information from the extended model. Therefore, Corfu comes with a ground software library and a reference implementation of ground software.

1.3 Original Contributions

In sum, Corfu enhances the state of the art and recent research with the following contributions:

- Corfu specifies a model specifically for small satellites, which relies on the YAML syntax. In contrast to the format of other frameworks, YAML requires less boilerplate code, which makes it easier to read and write for humans. Users can parse the YAML files with established parsers to process the model information in new ways by avoiding to introduce an utterly new syntax.
- To date, there is no full model for onboard software that covers all configuration parameters. Other model-based frameworks such as FPrime[10] still require configuration information in the source code. Having all configuration parameters directly available in the model in an easy parsable way like YAML facilitates the verification and further processing.
- In current approaches, the model for satellite software do not cover timing parameters. Corfu does include timing information for (periodic) thread execution, which allows early estimations of CPU usages and scheduling analyses. The same goes for memory usage.
- Introducing feedback from the user source code back to the model is a novelty in the field of the model-driven development for satellite software. Having information available from the source code enables improvements regarding performance and safety.
- Exploiting information from the (extended) model of the software at satellite operations is new. As a use case, we have implemented an event logging system to show positive effects. On the one side, executable files have to

store less constant data, and on the other side, log event messages require less bandwidth for transmission.

- With information from both models, engineered and extracted, we show that verification can be improved, e.g., checking stack sizes of threads.

1.4 Delimitation of the Work

The area of software frameworks for satellites is vast; therefore, we have some exclusions.

1.4.1 Small (Academic) Satellites

This work has been developed in the context of creating satellite software for small satellites, namely Cubesats. In those satellite projects, we relied on COTS, as most academic space missions do. Specifically, we had three types of microcontrollers in use in the projects running Corfu. Table 1.1 lists their specifications. As one can see, those microcontrollers are not very powerful. It is not possible to run extensive operating systems such as Linux on them. Therefore, we had to design the framework to run with minimal resources.

Microcontroller	Clock Frequency	RAM
Silabs EFR32FG12	40 MHz	256 KiB
STM32L4+	120 MHz	640 KiB
STM32F4	168 MHz	192 KiB

Table 1.1.: The specification of microcontrollers in projects for which Corfu was developed

The focus of our framework is to provide fast development of software for experimental satellites. Thus, we do not cover characteristics of bigger and commercial satellites, which come with additional requirements.

1.4.2 Extended Modeling-Driven Development

In the engineering field, there exists the term round-trip engineering (RTE), which synchronizes models with the source code[89]. In the one direction, that means if the model changes, the code will be re-generated. In the other direction, if the source code changes, the model will adapt to it. The primary goal of round-trip

engineering is to ensure that the model — and thus part of the documentation — is always consistent with the source code.

Our approach of extended model-driven development differs from round-trip engineering. Instead of keeping the same information up to date in both representations (model and code), Corfu combines information of both representations and uses them together. Another difference is that we use the (extended) model at compile-time and the software's run-time. Classical round-trip engineering only uses it at development time.

1.5 Partial Publications

Some results of the dissertation have already been published in journal and conference papers:

1. Frank Flederer, Sergio Montenegro: *Model-Based Framework for On-Board-Software*, Small Satellite Conference, 2021.

2. Frank Flederer, Sergio Montenegro: *A Configurable Framework for Satellite Software*, IEEE International Conference on Software Engineering and Service Science, 2021.

3. Accepted[1]: Frank Flederer, Sergio Montenegro: *Communication Middleware for Onboard Software in Space: A Survey of Techniques and Implementation Aspects*, 4S Symposium, 2020.

4. Frank Flederer, Sergio Montenegro: *A model-driven framework for small satellite software with feedback through static analysis of user code*, Pico and Nano Satellite Workshop, 2021.

5. Frank Flederer, Sergio Montenegro: *CORFU - Ein modellgestütztes Framework für On-Board-Software mit statischer Quellcode-Analyse*, Deutscher Luft- und Raumfahrtkongress, 2021.

6. Frank Flederer, Ludwig Ostermayer, Dietmar Seipel, Sergio Montenegro: *Source Code Verification for Embedded Systems using Prolog*, Workshop on Logic Programming, 2017.

[1] but not presented because the organizers canceled the symposium due to the COVID-19 pandemic.

7. Benjamin Grzesik, Tom Baumann, Thomas Walter, Frank Flederer, Felix Sittner, Erik Dilger, Simon Gläsner, Jan-Luca Kirchler, Marvyn Tedsen, Sergio Montenegro, Enrico Stoll: *InnoCube - A Wireless Satellite Platform to Demonstrate Innovative Technologies*, Aerospace, 2021.

Several parts of this dissertation have been adapted from those published papers (mostly in modified form). The overview of existing middleware systems (section 4.3) is adapted from the paper 3. The chapter 6 contains information of the related work sections from the papers 1, 2, 4, and 5. The figures 8.3 of development process and its description is adapted from the papers 2, 4, and 5. The figures 8.4, and 8.5 and their description texts are adapted from the paper 2. The figures 8.6, 8.7, 8.8, 8.9, 14.1 about the generated code and their description texts are adapted from the paper 1. The configuration examples from the sections 11.3 and 11.4 and their descriptions are also adapted from the paper 1. The description about Clang's abstract syntax tree in section 12.3.1 is adapted from the paper 6. The implementation and evaluation of the logging event use case (section 8.4.5) are adapted from the papers 2 and 5. The aspects of middleware implementations (appendix A) are adapted from the paper 4. The description about innocube (section 17.3) is adapted from the paper 1. The description of Corfu's tools and libraries in chapter 10 is adapted from the papers 1 and 2. The description of Corfu's test environment in chapter 14 is also adapted from the paper 1. Some information of the chapter 9 is adapted from paper 5 Implementation and evaluation details of the source code feedback for the log events system (sections 12.3.2, 15, and figures 15.1, 15.2) are also adapted from paper 5.

1.6 Document Organization

This dissertation consists of six main parts:

1. The introduction gives a short overview of the topic. It presents the motivation and a summary of the work.

2. The fundamentals part provides the necessary knowledge for this work. The fundamental part includes the characteristic of satellite missions and onboard software, its structure and development, and aspects of safety-critical software. In addition, the part gives an overview of model-driven development, which is the core development concept for Corfu. Finally, the part closes with the state of the art and presents other satellite software frameworks.

3. The third part presents the design of Corfu. First, it describes the development methodology and the fundamental conception with requirements, use cases, and the development process. In addition, this part describes the engineering model for onboard software, which Corfu uses. Finally, it outlines the organization of Corfu's software suite.

4. The fourth part presents realization aspects of Corfu. It shows the structure of configuration files and the generated code. In addition, the part presents verification steps and how Corfu accomplishes the feedback from user code to the model. The part ends with a description of automatic code testing.

5. The fifth part evaluates the implementation in regards of resource usage and process improvements. It describes how the usage of code feedback influences the software and the development process.

6. The conclusion part gives a summary of the whole project and an outlook on future work.

1.7 Color Conventions Used in the Document

For the source code in the listings, we have used different background colors. They indicate whether the displayed code is

```
1 user code,
```

```
1 generated code, or
```

```
1 ready code coming with Corfu.
```

Part II

Fundamentals

2 Characteristics of Satellite Missions

Commonly, space missions usually follow the same organization and procedure, which this chapter describes. Those common characteristics lead to requirements that are specific to satellite missions and thus also to software frameworks.

2.1 Hierarchical Composition of Components

Developers and engineers never create satellites only on their own. Instead, they develop the onboard and the ground segments in parallel. Both share the same communication interface for commanding the satellite and receiving telemetry from the satellite. On the software level, this communication interface defines the structure and identifier of telecommands and telemetry packets. A telecommand represents an instruction for the satellite, for example, enabling a payload device. Therefore, telecommands come from the ground to the spacecraft. However, there might be some telecommands that the satellite triggers, such as timed telecommands that are due. Telemetry works in the opposite direction: it provides information about the current status of the satellite or payload data down from the satellite down to the ground.

Space missions can be described hierarchically in several layers, as figure 2.1 shows. They consist of two main parts, the ground segment and the space segment. The communication between the ground segment and the satellites(s) uses radio links, such as s-band or ham radio. Most satellite missions only use one spacecraft. However, some missions include several spacecraft combinations of landers and rovers or companion satellites. Those multiple spacecrafts communicate via radio as well.

Each satellite consists of different subsystems, which accomplish different tasks. For example, there are subsystems for power management and attitude control. All of those subsystems come with computers, which are connected by bussers. We call such computers *nodes*. Nodes execute several applications in parallel. Applications

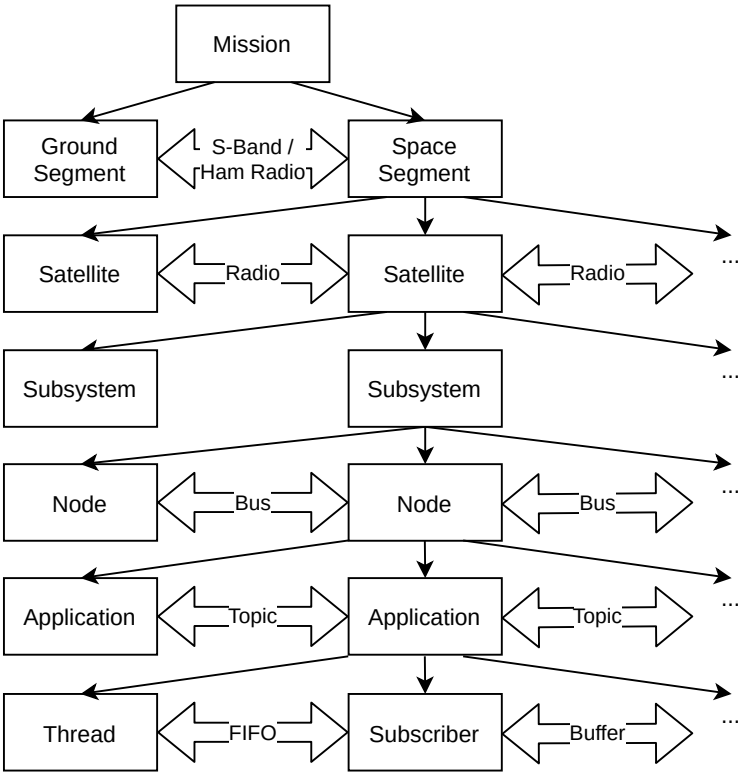


Figure 2.1.: Hierarchical composition of satellite components and their communication interfaces

are software modules that implement distinct features, such as managing timed commands or uploading and downloading data. In order to fulfill its task, applications execute threads and provide subscribers, which react to data and signals from other applications or to telecommands. Such primitives from the operating systems share memory directly and synchronize using synchronization primitives such as semaphores, FIFOs.

Let us formalize such satellite missions. They consist of the entities from figure 2.1 and their relations. Let Sat be the set of satellites in a mission and U be the set of subsystems. We define satellite configurations as

$$SC \subseteq Sat \times U \tag{2.1}$$

The subsystems consist of a number of computing nodes. A node represents computing nodes in a satellite; in other words, a satellite consists of a set of nodes. Nodes stand for the hardware (e.g., microcontroller) that executes the onboard software. Let N_s be the set of (computing) nodes on a satellite $s \in Sat$ and A be the set of applications. We define subsystem configurations as

$$UC_s \subseteq N_s \times A \tag{2.2}$$

Nodes need to communicate with each other using busses and networks that connect nodes. However, not every node can directly communicate with each other node. Let B_s be the set of busses on a satellite $s \in Sat$. We define bus configurations as the set of node sets being connected via a bus.

$$BC_s \subseteq \mathcal{P}(N_s) \times \mathbb{N} \tag{2.3}$$

The last element contains the number of physical busses. If a bus connection is redundant, its value is greater than one. On those bus connections, the software communicates with high-level concepts, such as virtual communication channels (also referred to topics). Applications can publish data on topics and subscribe to topics. Let To be the set of topics. Therefore, we define a topic configuration as

$$ToC \subseteq To \times \mathcal{P}(A) \times \mathcal{P}(A) \tag{2.4}$$

$$= \{(t, P, S) | t \in To \wedge P = \{a | publishes(t, a)\} \wedge S = \{a | subscribes(t, a)\}\} \tag{2.5}$$

Finally, applications consist of different elements. On the one side, we have active elements, which drive application execution. Those are threads, which come with own execution paths. In their execution, active elements make use of passive elements. For example, publishing data to topics, i.e., sending data, invokes passive

elements such as topics and subscribers. Let Th be the set of threads. We define application configurations as

$$AC \subseteq A \times Th \tag{2.6}$$

These are the basic concepts of satellite missions. Indeed, it is common to implement more advanced concepts on top of those, e.g., telecommands, their distribution, and handling.

2.2 Phases of Satellite Projects

Not only the actual flight but also the whole project of a space mission consists of different phases. The ECSS defined different phases that space missions shall follow[29].

- Phase 0: Mission Analysis / Need Identification
- Phase A: Feasibility
- Phase B: Preliminary Definition
- Phase C: Detailed Definition
- Phase D: Production / Ground Qualification Testing
- Phase E: Utilization
- Phase F: Disposal Phase

This approach usually knows only one direction forward through all the phases, following the waterfall approach. Small teams might handle projects better in agile approaches. The ECSS also provides a document for employ agile development for space software[31]. See section 1.1.4 for more information about agile development in satellite projects.

As one can see, the actual mission flight takes place in the last two phases, E and F. These are also the phases in which require the software to work reliably. The flight time can also be divided further, defining different requirements for the spacecraft and, hence, for the onboard software. Jens Eickhoff summarized those phases[32]:

- Pre-launch Phase: final checks, configuration of the software

- Launch and Early Orbit Phase (LEOP): disconnecting from the launcher, initiating attitude control, orbit correction maneuvers
- Commissioning Phase: performing operations tests, preparing payloads for operation
- Nominal Operations Phase: performing payload operations, daily work (orbit correction)
- End-of-life Disposal Phase: controlled de-orbiting

Onboard software has to cover all these in-flight phases. Every phase has different requirements for the spacecraft. In the commissioning and LEOP, only a limited energy budget is available because the solar panels are not deployed and calibrated yet. During those phases, there is no or only late communication with the ground station possible. Therefore, the software has to accomplish those phases autonomously. Not all processes in the software should run in every phase. Most payload processes should not run before the commissioning or nominal operation phase.

2.3 Small Satellites

We can classify satellites in different ways. Sweeting divides satellites based on their mass[92]. Mauro et al. refined his classification and extended it by medium, pico, and femto satellites[62]. Tab. 2.1 shows the classification. We can see that the small satellite includes every class with less mass. That means mini, micro, nano, pico, and femto satellites also belong to the class of small satellites.

A particular class of satellites is CubeSat. Their base is a cube with the size of $100 \times 100 \times 100\,\mathrm{mm}^3$[40] and a mass of maximum $1.33\,\mathrm{kg}$[16]. These values represent

Satellite Class	Mass in kg
large	$\geq 1,000$
medium	≥ 500 and $< 1,000$
small	< 500
mini	≥ 100 and < 500
micro	≥ 10 and < 100
nano	≥ 1 and < 10
pico	≥ 0.1 and < 1
femto	< 0.1

Table 2.1.: Classification of satellites based on mass[62]

the values for 1U. Cubesats can be bigger than just 1U; 3U and 6U are also common, which have proportional greater dimensions. For example, 3U satellites have a dimension of $300 \times 100 \times 100\,\mathrm{mm}^3$ and a mass up to $4\,\mathrm{kg}$.

In recent decades, satellites have become more and more accessible[93]. Due to the improved performance and quality of COTS, such components have arrived in space. Components-off-the-shelf are small and cheap, enabling low-cost satellites, which can be built even by amateur radio groups and universities. Davoli et al. estimate costs of $100,000 to $200,000 for building and launching a small low earth orbit satellite[27].

Not only differ small satellites in the selection of components but also the management approach is different[93]. The development of small satellites (especially in the academic field) follows a more agile approach than classical satellite project management. Engineers have to select the components wisely. Most components-off-the-shelf are not primarily designed for use in space; therefore, engineers have to evaluate thoroughly. Such evaluation includes tests that cover specific requirements such as radiation robustness. Producers might change and modify components-off-the-shelf during their production time, e.g., introducing new revisions or changing material or production processes. Therefore, it is crucial to work with components of the same lot. On the other side, components-off-the-shelf have proven to provide high reliability due to their massive use[93].

In contrast to the classical development of big satellites, developing with components-off-the-shelf requires more agility. If some components do not work in the project or different properties in different lots are detected, the team has to adapt the project and the process to such new information.

Such agile development approaches have to be supported by the tools used. The same is true for software development, which supports and welcomes changes and iterative approaches — as Corfu does.

3 Basics of Model-Driven Software Development

Model-driven development is an engineering approach, which has existed since the 1980s[82]. In classical development, the software is created based on documents. Those documents could be lists of requirements or — in agile developments — user stories. However, there can be a lot of documents, which partly also contain redundant information. Consequently, every time something changes, people have to make sure to update all involved documents manually. This approach can be very cumbersome and become expensive[36]. Model-based and model-driven development tackle such problems. Instead of having a lot of informal documents, MDD comes with a formal way to define a (software) system: models. Such models do not contain redundant information. Due to their formal style, it is possible to automatically process information from the model to accomplish verification tasks and auto-generate other artifacts, e.g., source code.

3.1 Process of Model-Driven Development

As Section 1.1.3 outlines, MDD sets itself apart from MBD by requiring automatic conversion from the model(s) to the code. Therefore, software structure and behavior information know one direction from the abstract representation to the software implementation. Figure 3.1 depicts the information flow. The basis is requirements, a textual description of features, behavior, and constraints of systems and software[102]. Engineers have to translate those requirements into a formal description. They can either describe them directly in the code, i.e., implement the code according to the requirements, or formalize requirements in another formal way, such as models. Model-driven development uses models to generate code eventually. Therefore, creating models does not only help for documentation, verification, and communication between stakeholders; it also directly expedites the development process.

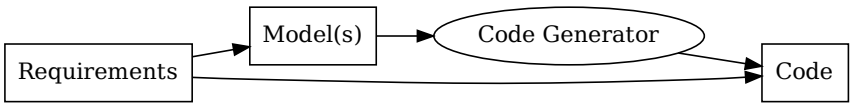


Figure 3.1.: Information flow in model-driven development

3.2 Increasing Abstraction Level

In the history of software development, designing and implementing software has become more and more abstract, and on the other side, reusability has been improved[64]. Figure 3.2 shows how the abstraction of software development has evolved. Each step introduces more abstract concepts and translates them into more primitive constructs.

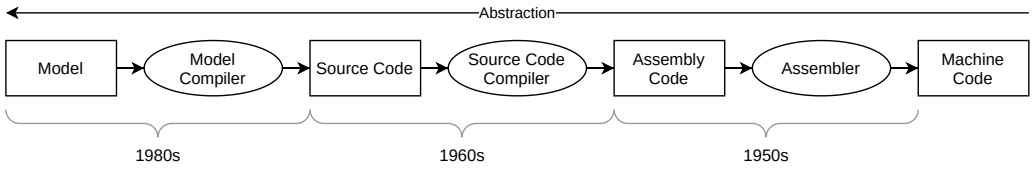


Figure 3.2.: Abstraction in software development (adapted from [64])

In machine code, developers must write their code directly as numerical values (e.g., hexadecimal code). The machine code encodes instructions and data together, which makes it hard to handle. In 1949, the first assembler code was developed and was established in the 1960s. It introduces commands for processor instructions, consisting of a short command and the parameters for instructions. [1]

The next step was high-level languages. Their development began with Fortran, which Backues released in 1957[4]. With the emergence of high-level languages, new programming concepts became easily accessible, which developers would have to encode into assembly code manually. For example, high-level programming languages come with loops. In order to implement loop behavior in assembly, developers have to combine conditional and unconditional jumps.

In high-level programming languages, different programming paradigms have emerged, which comes with further abstractions. Very well known is the object-oriented programming paradigm, which ties data to functions and comes with inher-

[1] Even if assembly code looks primitive nowadays, the apollo mission shows that complexity was still manageable. The MIT Instrumentation Laboratory implemented a dynamic real-time scheduler to execute multiple tasks quasi-parallel in assembler code.[39]

itance. Besides, there are the programming paradigms of functional programming and logic programming, which approach development differently. Depending on the problem domain, one paradigm might be better suitable than another. Therefore, some programming languages support multiple programming paradigms.

Another development of high-level programming languages is domain-specific languages (DSLs). They come with keywords and concepts tailored to problem domains, distinguishing them from general-purpose programming languages. Models are the last step in our diagram. Similar to high-level programming languages, they describe the structure and behavior of the software. However, they are abstract in order to keep the focus on specific aspects of the software. Also here, there exist general-purpose and domain-specific models (e.g., domain-specific modeling languages). Technically, models are not the end of the line. Models can also be described in another language: the metamodels.

3.3 Elements of Model-Driven Development

There are different artifacts and tools are used for MDD. This section gives a short introduction to those elements.

3.3.1 Systems

Favre identified three types of systems[35]:

- physical systems,
- digital systems, and
- abstract systems.

Models can describe all of those system types.

Physical models reflect the behavior of physical objects. For example, the satellite structure is a physical object. For satellite missions, engineers use physical models to investigate the behavior of satellites, e.g., the thermal flow and its movement. Hence, physical models are an integral part of satellite development because engineers can not fully replicate the orbit's physical environment on the ground. Therefore, engineers often have to develop against a physical model of the satellite and the target orbit for satellite development. For example, this includes the development and tuning of the attitude control system.

Favre characterizes digital systems as the content of digital memory, basically software. Such digital models are those that software engineers build. In satellite missions, digital systems are all software artifacts that run on the satellite and the ground. The structure of digital models depends on the software architecture and the domain in which the software runs.

Finally, abstract systems are neither physical objects nor digital systems — they are concepts and approaches, which have (not) materialized into the real world. These could be business rules, which define processes, which humans perform. In satellite missions, manuals for the operating staff are an example of an abstract system.

3.3.2 Models

The word "model" is used in many disciplines and, therefore, comes with many definitions and descriptions[8, 12, 49, 85]. In model-driven development, models help to describe a system clearly and concisely. For the field of model-driven development, we define models as follows.

Definition 3.1. A model M in the context of model-driven development is a collection of information describing the structure and behavior of real or abstract systems or both. It does not incorporate every detail of a system because models focus on specific parts of the system.

The literature knows two categories of models: descriptive and specification models[85]. They differ in the source of construction. Descriptive models are created from existing systems. For example, such models can describe a physical system or existing software. Specification models are created from the specification or requirements. In this case, the system to be modeled does not exist yet. Instead, the system will be built based on specification models. Hence, specification models are such models that model-driven development uses because they first require defining models and create the system from those models.

In [64], Mellor et al. establish four points that make good models:

- Good models contain only such information that is needed in order to describe a problem domain. All other information should be omitted in order to keep focus on the issue.[2]

[2]Indeed, it is good practice to have several models describing different problems, which contain different types of information.

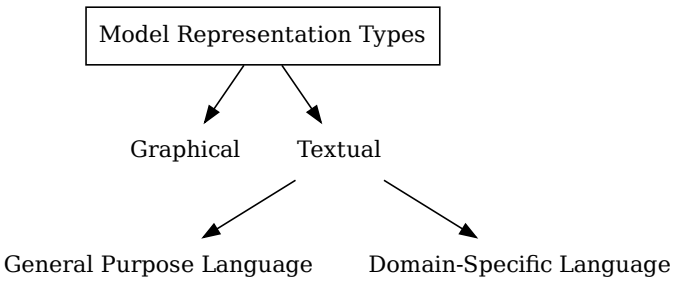


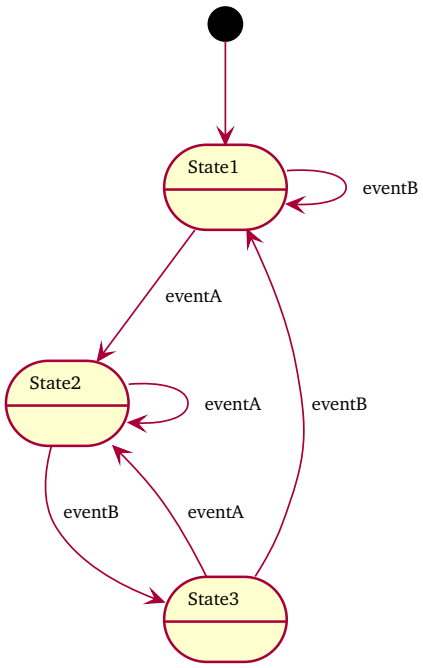
Figure 3.3.: Representation types of models

- Good models contain all types of information that are required to describe the problem domain accurately. It should approximate the behavior or structure of the real or abstract object as close as possible.
- Good models facilitate the development. They should be easier to create than the final artifact. By operating verification and tests already on the model improves the quality of the final product.
- Good models are easy to grasp. This improves the communication between the stakeholders and reveals misunderstandings earlier.

One can treat any entity or artifact as a model. If we have a look back at figure 3.2, not only the model left is a model in the broader sense, also source, assembly, and machine code are representations of the same software.

There is no limit to the number of models that might describe the same system. Models can concentrate on specific aspects of the software. In order to concentrate on specific aspects of the system, models can leave out unnecessary information for the focus. Omitting unnecessary details allows developers to concentrate on specific aspects of the software. For example, some models describe the static structures of software, and some describe the software's dynamic behavior.

Apart from the type of information that models contain, models also use different representation forms, which figure 3.3 shows. For example, UML uses graphical representations to describe different types of models. UML is a collection of several types of diagrams. Version 2.3 of UML knows 14 different diagrams. Those diagrams can be divided into structural and behavioral diagrams. Class, component, and deployment diagrams are examples for structural diagrams and activity, use-case and sequence diagrams are examples for behavioral diagrams.



(a) Example: Graphical Model Representation (UML)

```
1  byte state = 1;
2
3  proctype eventA() {
4   state = 2
5  }
6
7  proctype eventB() {
8   if
9   :: (state == 2) -> state = 3
10  :: (state != 2) -> state = 1
11  fi
12 }
```

(b) Example: Textual Model Representation (PROMELA)

Figure 3.4.: Examples of graphical and textual model representations

Another format is textual representations of models. There are two types of textual representations: 1. applying a standard markup language (e.g. XML, YAML) and 2. defining a domain specific modeling language (DSML). In comparison, both have advantages and disadvantages. Using general markup languages facilitates processing model data in other tools because parsers usually exist for many programming languages. DSMLs, on the other hand, come with their own syntax and semantics, which perfectly cover the problem domain. By providing appropriate concepts and syntactic constructs, DSMLs facilitate expressing domain-specific solutions concisely and efficiently. They exploit that domains already have given nomenclature and concepts, which they incorporate.

Figure 3.4 shows two examples of model representations. The left one is a UML state machine diagram, the right one is PROMELA[41], which both describe behavioral aspects of the software.

As we have mentioned earlier, many models, which cover different aspects, might map a system. For example, such aspects could be the class structure and their relations or the behavior of function calls. We call those different aspects *model classes*.

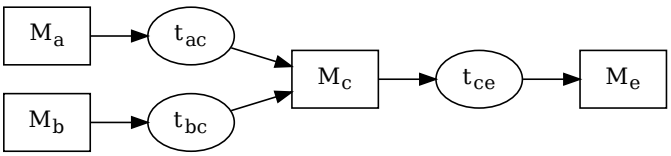


Figure 3.5.: Example of model transformations in a software project

Definition 3.2. A model class $\mathcal{C}$ represents a type of model, which restricts its content to a given part of a system. Model classes are independent of a system's implementation details; instead, users can use them for similar systems.

For example, class diagrams are model classes that engineers can use for all software systems. Each model M is part of a model class.

$$M \in \mathcal{C} \tag{3.1}$$

There are two types of model classes: abstract and executable[64] model classes. Executable model classes contain all the information that is required to run the system. In the field of software engineering, this is usually the source code, which is the special model class C_e.

3.3.3 Model Transformation

When implementing software in model-based development, using models for reference suffices. However, in model-driven development, this is not enough because model-driven development requires automatic transformation from models toward software implementation. Transformability does not mean that each model should directly be translatable into source code; it is also possible to transform models into intermediate models, which finally are the basis to generate code (see figure 3.5). The figure has two initial models, M_a and M_b, which are transformed into one common model M_c. Hence, such intermediate models can consolidate information from several models into one. The output of the overall process, M_e (e.g., executable), transitively depends on the two input models, but the direct transformation into M_e has the intermediate model M_c as the source.

Let $\mathcal{C}_a, \mathcal{C}_b$ be two model classes; we express model transformations of $\mathcal{C}_a$ models to $\mathcal{C}_b$ models as follows.

$$t_{ab} : \mathcal{C}_a \rightarrow \mathcal{C}_b \tag{3.2}$$

Model classes might come with their own information independent of the given system. For example, a model covering sensor and actuator drivers (i.e., the interface towards the physical world) might also comprise physical laws. However, this additional information might be irrelevant for other model classes. Hence, two models might share information, but they are not necessarily sub-/supersets in one direction.. Let M_a and M_b two different models in the two different model classes $\mathcal{C}_a$ and $\mathcal{C}_b$.

$$M_a \in \mathcal{C}_a, M_b \in \mathcal{C}_b : M_a \subseteq M_b \wedge M_a \supseteq M_b \tag{3.3}$$

4 Structure of On-Board Software for Small Satellites

This chapter gives an overview of the general structure of satellite software. It addresses common parts of onboard software such as operating systems, communication middleware, common applications, and telemetry.

4.1 Basic Software Structure

The basic software stack for spacecraft is similar for most projects – see figure 4.1. At the low part, there is an operating system[1]. Its purpose is to abstract the hardware from the upper parts of the software stack. Additionally, it usually provides real-time scheduling for threads. The board support package (BSP) is the part of the operating system that contains hardware-specific code. For each new hardware device supported by the operating system, developers have to implement such a board support package.

The operating system itself just provides basic features, such as thread management and some inter-thread communication primitives. In order to use more advanced mechanisms, satellite projects usually integrate a framework, which has been developed specifically for spacecraft software. For an overview of existing software, see section 6.

The next part is the communication middleware. In most cases, the communication middleware is not an extra software; instead, it is often part of the framework. Compared to primitive communication mechanisms provided by the operating system, a communication middleware usually comes with more complex and more flexible ways of communication between different applications — not only within one computing node but also across different nodes. See section 4.3 for an overview of different aspects and techniques of communication middleware implementations.

[1] Even in the very early stages of astronautics, NASA developed several priority-based operating systems, such as EXEX, which they developed in the Apollo program[32].

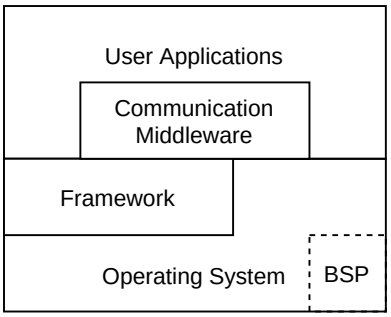


Figure 4.1.: A common software composition for satellites

The most up-level part of onboard software is the user applications. There, users implement the mission-specific logic. The user code uses the lower-level parts: the communication middleware for flexible communication between applications, the framework for generic spacecraft software features, and the operating system for hardware access.

4.2 Operating Systems for Satellite Software

Developers might implement some small microcontrollers on bare-metal, but most computing nodes in spacecrafts use an operating system for hardware abstraction and task scheduling. In the context of satellite software, operating systems have two primary purposes: abstraction of the hardware and process management[24], which this chapter shows. Additionally, we present different available operating systems for space usage.

4.2.1 Hardware Abstraction

Operating systems come with an application programming interface (API) independent of the underlying hardware. Some of them follow established interfaces, such as the Portable Operating System Interface (POSIX)[44]. The implementation of the API is often hardware-dependent. Hence, there exist different implementations depending on the hardware devices that the OS supports.

4.2.2 Process Management

Usually, computing nodes in spacecrafts have to several accomplished tasks simultaneously. Operating systems provide different mechanisms to achieve such behavior. This section provides definitions of different terms and concepts.

Processes and Threads

Processes represent independent programs that run (quasi) parallel. Each process has its own memory space, which requires a memory management unit (MMU). Additionally, processes have their own register set, which the operating system's scheduler saves and restores every time it preempts a thread. Some definitions of threads call them "lightweight processes" [95]. Each thread also has its own set of registers, which the scheduler manages. However, multiple threads may belong to a process; thus, they can access the same memory area. Some operating systems, e.g., RTEMS, do not support MMUs. Consequently, they provide only one global process; threads execute the parallel computing paths.

Co-Routines

Usually, developers do not have to care about preemption when designing processes and threads. They are programmed just like they do not have to share the processor with other threads. Usually, they have an endless loop for their program flow. The scheduler takes care of revoking the processor of a running thread and reassigning it to other threads.

Co-routines, on the other hand, are aware of the other threads. They deliberately give control to the scheduler in order to give other threads a chance to run[51]. This behavior enables co-routines to run with non-preemptive (cooperative) schedulers. Another difference between processes and threads is that all co-routines share one stack.

Synchronization

The different threads and processes running on an operating system usually do not perform their work independently. Often, they need to share data between them or to synchronize with each other. The OS usually provides some standard mechanisms[86]:

Mutex There might be portions of code, which the scheduler shall not execute in parallel. One example is code that accesses a hardware device, e.g., writing data to a UART. The software creates a mutex object or handler to access such devices, which every piece of code uses that accesses the device (critical section). Before the device is accessed, the code must enter the mutex and leave it afterward by calling appropriate methods or functions. When developers apply object-orientated programming, it is sensible to use a guard object here (see 17.1.1).

Semaphore Software uses semaphores to send signals between threads or processes. Both threads hold the same semaphore object or handler. A thread might call a method or function to suspend until the semaphore forwards a signal. Other threads can signal the waiting threads to resume, also by calling a method or a function. In contrast to mutexes, semaphores also could have a counter, which counts up each time a thread waits for a semaphore and counts down each time a thread sends a signal. Consequently, it is possible to implement a limitation on the number of threads for specific code sections.

FIFO When threads exchange data locally, they can use semaphores and shared memory addresses. However, this limits the number of data packets to one. The FIFO provides a fixed number of data packets, which can be filled and retrieved by invoking methods or functions. For the synchronization of threads or processes, there usually exist blocking FIFOs. A blocking FIFO suspends a thread that wants to read from an empty FIFO until new data is available (or the reading reaches a timeout). On the other end, if a thread inserts data into a FIFO that is already full, the thread is suspended as well.

More advanced synchronization and communication strategies exist on some operating systems, e.g., some provide a communication middleware. For more information about this, see section 4.3.

4.2.3 Process Scheduling

Real-time systems use two types of scheduling in real-time systems: the a priori static scheduler and the a posteriori dynamic scheduler.

Static Scheduling

Static scheduling provides maximal predictability about the execution times of tasks. Engineers specify time slots in which the tasks run already before compilation. They assign each task to specific time slots, which will not vary at run-time. For handing (unplanned) interrupt tasks, a certain amount of time slots must be reserved, according to the maximal occurrence rate. Due to the rigid scheduling, it is not possible to adapt reaction times based on actual inputs.

Dynamic Scheduling

In dynamic scheduling, there are no time slots with predetermined allocation. Instead, at run-time, a scheduler selects the next task to run (continue). The selection process follows a known algorithm. For many real-time operating systems, the scheduling mechanism uses fixed priorities. Not changing the priority of tasks all the time increases the predictability of the reaction behavior. Usually, real-time systems always select the task that is ready with the highest priority. When two tasks have the same priority assigned, some operating systems, e.g., Rodos and RTEMS, apply a round-robin scheduling, i.e., schedule the tasks alternatively with identical time slots. Round-robin scheduling is only possible when the scheduling is preemptive, i.e., the scheduler can suspend running tasks in order to resume them at a later time point. Preemption is very suitable for dynamic real-time scheduling. If a task with high priority becomes ready, the scheduler can suspend lower priority tasks to finish the high priority one first.

4.2.4 Resource Management

Apart from processor time management in the scheduler, operating systems usually also manage other resources on a system. Memory is another resource that operating systems manage. They assign the required memory to the different software components. In safety-critical systems, the software shall not use dynamic memory (cf. section 5.1.1); therefore, such operating systems allocate the memory only in the start-up phase.

Another part of the management is the protection of resources. For the memory, operating systems might implement memory protection to prevent software components from accessing other memory sections.

The same also goes for other resources in a system, such as hardware devices. Operating systems can manage those devices and assign them to the different software components on request. In some cases, they ensure that such devices are not accessed simultaneously by applying synchronization mechanisms (see section 4.2.2.

4.2.5 Overview of Existing Operating Systems

Several real-time operating systems have already been used in satellites projects, which we shortly introduce in this section.

Rodos

In the strict sense, Rodos (Real-time On-board Dependable Operating System) is more than an open-source operating system; it provides extended, high-level features, such as a communication middleware[69]. Its development started at DLR and continued the development and maintenance at the University of Würzburg. It is available under open source conditions[2].

Rodos comes with a fixed priority scheduler for running bare-metal on the different ported hardware, including several ARM boards (Raspberry Pi, STM32, Smartfusion2). Additionally, Rodos can run as a guest on other operating systems, such as Linux, Posix, and MacOSX. In this case, it reuses the tasking/threading mechanisms of the underlying operating system. Developers program satellite software against a hardware abstraction layer, which makes the software agnostic of the hardware. Having an abstraction layer is beneficial for development and testing, i.e., the software can be developed and tested on a desktop computer running Linux, and afterward, it can run on the actual hardware. Indeed, further testing of the hardware functionality and timing measurements have to be done on the actual hardware because both might differ from the desktop hardware.

The communication middleware provides a loosely coupled communication between applications in the software. It follows the publish/subscribe principle, i.e., applications might subscribe to a topic and get notified every time an application publishes new data to the topic. This data exchange is not limited to the software on one computing node. Rodos comes with a gateway service, which enables to forward topic messages across several computing nodes. In this case, the involved computing

[2]https://gitlab.com/rodos/rodos

nodes have to be connected via a network or bus. The nodes could also be connected indirectly; if two nodes have no direct connection but a common neighbor, the middleware can route topic messages through the common neighbor node.

In contrast to other approaches, e.g., those that follow the POSIX standard, Rodos forbids creating and destroying threads dynamically. Instead, it is common practice to instantiate all threads and topics statically. The set of threads is always constat; this highly improves timing predictability because the set of threads is always constant.

RTEMS

RTEMS (Real-Time Executive for Multiprocessor Systems) is an open source[3] operating system developed by the OAR Corporation. Towards the spacecraft software, it provides a POSIX 1003b API. It also implements a fixed-priority scheduling mechanism with a configurable round-robin procedure. In contrast to other POSIX operating systems, RTEMS does not provide multiple processes; instead, each software runs in one global process but in different threads. The difference is that the memory context is the same for all threads; there is no memory protection across the different threads. Like other operating systems, RTEMS comes with simple thread synchronization features such as mutexes, semaphores, and signals.

Salvo

Pumpkin Inc. provides a CubeSat Kit, which delivers the base structure for cube sat missions — this includes a hardware and software foundation. The core of the software, which comes with the kit, is the real-time operating system Salvo. Pumpkin Inc. did not primarily develop Salvo for space application, but several space projects had already applied it.

In contrast to many other real-time operating systems, Salvo does not provide preemption. Instead, it relies on cooperation between the tasks. That means that every task should step back from occupying the CPU to give other tasks the chance to progress. One advantage of this approach is that it is not required to hold a separate stack for each task. Instead, all tasks share one stack.

However, task synchronization, for example, via semaphores, is still required because tasks might release the CPU while holding exclusive access to some device. Moreover,

[3] https://www.rtems.org/

it provides communication concepts such as message queues, which threads can use to exchange data. The developers have written Salvo in pure C and, therefore, also provides a structural API for flight software.

FreeRTOS

This open source[4] real-time operating system dates from embedded devices, such as IoT. However, several satellite projects use it. Like Salvo, it uses C as the programming language and, hence, FreeRTOS' API towards user software follows the structural programming paradigm. It supports both preemptive task scheduling and co-routines, cooperative tasks voluntarily releasing the CPU. Co-routines share the same stack. FreeRTOS provides standard communication primitives such as semaphores and queues. In its plus packet, which is not freely available, FreeRTOS comes with some already implemented features, such as TCP and UDP stacks or an SSL implementation.

VxWorks

VxWorks is a real-time operating system from Wind River Systems[83]. Interplanetary missions such as NASA's Pathfinder mars mission rely on VxWorks as OS. VxWorks is proprietary and not freely available. It comes with a real-time kernel and many features like dynamic linking, execution of containers for virtualization. VxWorks implements a priority-based and preemptive microkernel[54]. Towards applications, VxWorks provides a POSIX interface, which also entails synchronization mechanisms and IPC. Many different embedded systems use VxWorks, such as robotics, cameras, and also satellites. In the latest version, VxWorks also comes with a stack for graphical user interfaces[106].

4.3 Communication Middleware for Satellite Software

Operating systems already come with primitive communication features for exchanging data between different software components. However, they have their limitation. For example, it is not possible to send data between different computing nodes. Communication middlewares provide advanced technologies for exchanging data between software components and even between nodes.

[4] https://freertos.org/

4.3.1 Services Concept

There are several approaches to standardize communication between space and ground segments and for onboard communication and data exchange between different spacecrafts. Often, the communication standard influences the design of the software. For example, the Packet Utilization Standard addresses services in the onboard software, suggesting a software design dividing into predefined applications.

4.3.2 Publish/Subscribe Principle

The publish/subscribe principle is a typical communication pattern used in many middleware implementations. It provides a loosely coupled message exchange between applications. The software instantiates one or more communication channels, also called topics, which publishing and subscribing applications use for exchanging messages.

4.3.3 Existing Middleware

Several concepts for communication middleware for both intra-node and inter-node communication are available. Some of them are also freely available under open-source conditions. We present a selection in the appendix A.1.

4.4 General Entity Types of On-board Software

Onboard software consists of different components, which perform different tasks. We have identified three component types, which this section presents: applications, shared resources, and communication channels (topics).

4.4.1 Applications

Applications are the active drivers of spacecraft software. These are the only items that contain threads, i.e., active code execution paths. They might work independently, but they communicate with other parts of the software in most cases. For this, they define a straightforward communication interface, which constitutes how

data can be passed to the applications and what data the application is ought to send to the outside.

4.4.2 Shared Resources

Shared Resources are the only elements in the source code that several applications might share. Hence, each module is a single instance of a class. They are instantiated at the node level and passed accordingly via dependency injection into the applications. The main difference between shared resources and application is that applications have active execution paths, i.e., threads, whereas shared resources are passive objects whose methods are called by applications. Due to their nature of being used by several applications, semaphores have to protect their access. This can be ensured e.g. by the `ThreadSafe` design pattern (see section 17.1.1).

4.4.3 Communication Channels (Topics)

Most frameworks provide middleware for communication between separate software parts, e.g., applications. In contrast to pure method calls, such systems provide a loosely coupled message exchange. Additionally, communication middlewares come with flexible communication paths with dynamic entrance and exit of applications. They provide different communication patterns, such as the publish/subscribe principle.

4.5 Classification of Applications

The set of all software applications that are part of a satellite system could be classified differently. This section presents those various ways of set partition.

4.5.1 Responsibility for the System

The software applications' responsibilities cover different parts of the space missions.

Node external communication (network, bus, sensor, actuator), communication middleware

Satellite guidance and navigation, payload/scientific, mode manager

Mission path planning

Every computing node on the satellite requires the applications on the node level. Those applications are necessary to make the software work together with other nodes. Consequently, those applications must be instantiated on each node there.

The satellite applications are required for the satellite to work in general, but it might not be important on which computing node the application runs. Thus, it suffices that they run on only one of the nodes in the satellite. The applications on the node level ensure that the spacecraft (and mission) applications can interact with each other (transparently). In addition, those higher-level applications might communicate with node-level applications, but only with the local ones. The node applications of other nodes are not directly accessible. However, this should not be a problem because all instances of those applications provide the same service.

The third group is the set of applications required once within the whole mission, i.e., one instance is required for the whole mission, either on the ground or in space. This type of application also uses the services of node-level applications.

4.6 Common Topics

Before looking at the typical onboard software applications in the next section, we briefly overview common topics that such applications use. Those topics are common for almost all satellites, independent of the mission and payload.

TelecommandUplink Some satellites may use the Rodos middleware for onboard communication and the packet exchange between ground and space. In the latter case, the software uses the TelecommandUplink topic for sending telecommands up to the satellite. Usually, only the Uplink application subscribes to this topic to process and forward the telecommand messages to other nodes and applications.

TelemetryDownlink This is the counterpart to the TelecommandUplink. If the satellite uses Rodos topics for ground/space communication, the software uses this telecommand. Usually, only the Downlink application publishes data on this topic, and the ground station subscribes to it.

Telecommand The telecommand topic distributes telecommands from the uplink application to all applications. Every application checks the destination address and executes the telecommand if applicable.

Telemetry Applications publish their telemetry data on this topic. The downlink application subscribes to it, transforming it into the downlink packet format and sending it down to the ground. The topic's data structure already represents generic topic packages, which do not distinguish between the standard and extended telemetry.

Anomaly All applications that report anomalies use this topic to report anomalies. Usually, there is an application subscribing to this topic to store reported anomalies and provide more information to the operations crew.

ModeChange The onboard software goes through several modes depending on different factors, such as sensor values and mission phases. Whenever a software component (ModeManager application) decides to switch the mode, it announces the new mode on this topic. All applications that have to react to mode changes subscribe to this topic.

ThreadIsAlive Every periodic thread shall send an alive message for each iteration. They publish the alive message to this topic, which the Watchdog application subscribes.

4.7 Common Applications

This section presents some applications that are common to most satellite missions. They are either part of the node or spacecraft responsibility layer. Therefore, Corfu comes with reusable reference implementations of those applications. User can import them into their satellite projects and adapt them if necessary. The table 4.1 shows the classification of the standard applications into the responsibility layers.

This section does not list subscriptions of the `Telecommand` topic and the publication of the `Anomaly`, `Telemetry`, and `ThreadIsAlive` topics. They are obligatory for all applications listed in the following.

Node	Satellite
Anomaly Collector	Downlink
Boot Manager	Mode Manager
Housekeeper	Uplink
Redundancy Manager	
Timed Commands Manager	
Watchdog	

Table 4.1.: Classification of common applications into responsibility layers

4.7.1 Uplink

Usually, one or two (redundant) modems receive telecommands from the ground segment. The `Uplink` application accesses the receiving hardware and forwards it further to the local `Router`. Any integrity checks are usually already done at this level. If a message's data is not valid, the application immediately discards the message and reports an error to the `FDIR` application. On satellites with multiple computing nodes, it is common that only one of them provides an uplink. Consequently, the software needs to distribute telecommands to the other computing nodes.

Configuration	Access to the modem / driver
Input Topics	-/-
Output Topics	TelecommandUplink

4.7.2 Downlink

The `Downlink` application is the counterpart of the `Uplink`. The onboard software forwards every telecommand packet that applications generate on the satellite to the `Downlink`. The `Downlink` mainly has two responsibilities. On the one hand, it manages the sending modem, and on the other hand, it transforms the telemetry packets into the protocol used on the radio link. Converting telemetry packets to the radio protocol includes combination and segmentation of telemetry packets. On a satellite with multiple computing nodes, it is common that only one of them provides a downlink. Therefore, every computing node that generates telemetry for the ground segment must forward it to the computing node with the `Downlink` application.

Combining the `Uplink` and the `Downlink` into one app is not uncommon.

Configuration	Access to the modem / driver
Input Topics	TelemetryDownlink
Output Topics	-/-

4.7.3 Mode Manager

As already described in 2.2, a space flight undergoes different phases, also called modes. The onboard software is the driving force for applying and transforming between modes. Following our modular approach, the `ModeManager` is responsible for handling modes. Different events may lead to switching the operation mode, for example, reaching a particular point in time or location or reporting an anomaly. Those events are usually generated by other applications, for example, the `TimeManager` or the `LocationManager`. The events are shared via the middleware communication system with other applications, as can be seen in the figure 4.2.

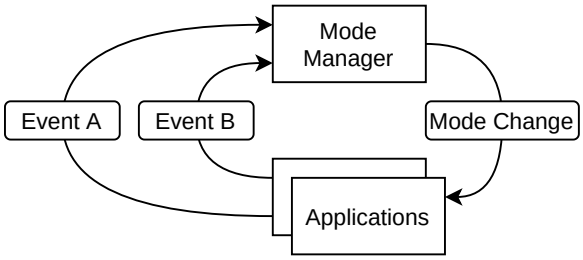


Figure 4.2.: The environment of the mode manager within the software

Every time the `ModeManager` receives an event, it checks whether the condition for a mode transition is fulfilled. If this is the case, it notifies all other applications about the new mode. In turn, every application is then responsible for changing its behavior according to the new mode. Changing the mode in the applications may include different behaviors: enabling/disabling threads or topic subscriptions, changing periodic interval times, or scheduling/canceling of timers. Consequently, every application has to know about all possible modes the software might enter. Alternatively, it could implement default behavior and implement only actions for those modes that require different behavior.

Configuration	Event/mode mapping
Input Topics	All the events
Output Topics	ModeChange

4.7.4 Anomaly Collector

The `AnomalyReporter` is a part of the FDIR concept. Its purpose is to inform the ground segment about onboard errors (anomalies). Whenever an application encounters unexpected situations, e.g., wrong commanding, it generates an anomaly report and forwards it to the `AnomalyReporter`. The `AnaomalyReporter` may also implement some cross-application recovery mechanisms to react to reported anomalies.

Configuration	-/-
Input Topics	Anomaly
Output Topics	-/-

4.7.5 Boot Manager

The onboard software is not always set in stone (read-only). Often, the software provides a mechanism to update over the air. For safety reasons, the onboard software installs new versions in parallel to the current version. The `BootManagers` task is to manage the parallel installation of multiple software versions. Additionally, it configures the boot loader to load a new version on the next start. Having multiple software images in the memory and a configurable bootloader enables to switch between different software versions. The boot loader usually provides some safety features, such as falling back to a working software version if a new version does not start correctly. However, this is not the scope of the `BootManager`.

Configuration	Hardware booting information
Input Topics	-/-
Output Topics	-/-

4.7.6 Timed Commands Manager

Some commands are already scheduled in advance, i.e., they shall be executed at specific time points. This is what the `TimedCommandsManager` is for. Via telecommands, the ground segment can schedule telecommands and even sequences of commands to be executed in the future. Of course, the `TimedCommandsManager` also provides features to manage (e.g., delay) or remove scheduled telecommands.

Configuration	-/-
Input Topics	Telecommand
Output Topics	Telecommand

4.7.7 Watchdog

Every application should work separately. Consequently, blocking applications should not impede other applications. Therefore, each application is encouraged to regularly send a heartbeat to show that it is still alive. The `Watchdog` 's purpose is to listen to those signals and report if an application does not respond anymore. As a consequence, the watchdog could reset the stalled application.

Configuration	-/-
Input Topics	threadAmAlive
Output Topics	-/-

4.7.8 Housekeeper

In order to provide a quick overview of the current status of the spacecraft, every application should regularly report their most vital variables, which suffice to tell what is going on at each application. The purpose of the `Housekeeper` is to collect that variable information from each application and combine them into one (or a very few) telemetry packet(s). Finally, it sends that telemetry down to the ground segment.

Configuration	Layout of the standard telemetry
Input Topics	-/-
Output Topics	Telemetry

4.8 Telemetry

To gain information about the spacecraft's current state, the ground operator relies on different types of telemetry, real-time and additional telemetry.

4.8.1 Standard Telemetry

Without explicit requests by the operator, standard telemetry is autonomously sent by the spacecraft periodically. This is accomplished by the `Housekeeper` application (see 4.7.8). The `Housekeeper` periodically requests all the other applications for vital information, primarily central variables from the application. Selecting suitable values to be reported via standard telemetry is very important. On the one hand, the size of telemetry packets is limited. Depending on the number of applications on a system, applications might contribute only a few variables to the standard telemetry. On the other hand, the purpose of standard telemetry is a quick overview of the state and healthiness of the spacecraft. Therefore, essential variables are those, which immediately show when something goes wrong in a subsystem.

4.8.2 Extended Telemetry

Unlike standard telemetry, extended telemetry is not being collected and sent autonomously; instead, the ground has to request them explicitly. Requesting extended telemetry is usually achieved via predefined telecommands. The content of additional telemetry is of varying nature. Most applications provide telemetry that exposes more variable values than are provided by standard telemetry. Having more information allows the ground operator to investigate further issues that he or she discovered while observing standard telemetry.

4.9 Communication with Ground

In contrast to the communication onboard or between satellites flying in formation, the link between the space and the ground segment comes with special requirements.

The selected communication protocol has to meet those requirements. It is possible to develop a proprietary protocol. However, if third-party equipment, e.g., ground link antennas, is used, the communication usually has to be compliant with an established protocol standard, such as CCSDS Space Packet Protocol[18] or ECSS Packet Utilization Standard[30].

5 Safety-Critical Software

Onboard software is a safety-critical part of satellites because it takes control of the most vital parts of the satellites. The safety-critical part includes telemetry generation and telecommand handling, thermal and power management, and attitude control. A failure in such components might lead to a loss of the satellite. Therefore, system engineers have high expectations of fault detection, isolation, and recovery (FDIR) and the quality of the software.

The history shows that software bugs arise — even in safety-critical applications, as section 5.2 shows. There are common software constructs or features that are error-prone. Therefore, some coding standards have been established that provide rules to improve software dependability. This section presents several programming standards and coding conventions.

5.1 Standards and Code Conventions for Reliable Source Code

Developing safety-critical software is not new, and it is also not limited to satellite software. There are many areas where safety using safety-critical software. Consequently, standards and conventions for writing reliable software (also in C++) have been established.

5.1.1 The Power of 10

Holzmann from the Jet Propulsion Laboratory proposes ten rules for safety-critical software[42]. He mainly focuses on rules that tools can automatically test.

No abstrusive control structures The control flow should be easily visible. Hence, developers shall not use hard-to-follow control constructs such as arbitrary jumps (`goto`) and function pointer.

Fix iteration limits for loops It is necessary to let loops not run for an arbitrary amount of time to guarantee real-time properties. Instead, cap the number of iteration for each loop. The same goes for recursion, which might be called directly or indirectly for an unknown number of times.

No heap allocation after initialization The memory usage should remain constant after initialization. No memory shall be newly allocated or freed after initialization. Omitting dynamic memory avoids certain types of programming errors, such as memory leaks. Additionally, not using dynamic memory minimizes memory fragmentation.

Keep functions short To make functions as lean and testable as possible, they should not exceed 60 lines of code.

Use assertions It is common practice to check values by assertions, even situations which *should* not fail. Holzmann suggests that every function should contain at least two assertions.

Keep data locally When declaring a variable, its scope defines the amount of code that can access its data. To avoid unintended access, developers shall declare variables in the smallest possible and suitable scope.

Check input and output A function usually operates on a specific domain of its input parameters (incl. the return value). Hence, each function should check the validness of its received parameters. Additionally, each caller should check the return values of called functions. This information often contains hints of erroneous executions.

Use the preprocessor only for includes Instead of defining constant values, use static constexpr. In most cases, real macros are not necessary; better use (inline) functions instead. Inline functions improve debug-ability and avoid pitfalls when using macros, e.g., forgetting parenthesis for a macro's parameter.

Limit pointer usage Do not hide the pointer property behind some typedef. Additionally, developers should not use more than one pointer level, i.e., they should not use pointers to a pointer and so forth. Finally, developers should not use function pointers because it prohibits static analysis and might cause inadvertent recursion.

Enable all compiler warnings Use pedantic compiler settings for warnings. Consider every warning to be an error, i.e., avoid having compilation warnings at all.

5.1.2 Industry C++ Coding Standards

The aerospace and automotive industries produce much safety-critical software. In those sectors, there are usually collaborations between many different companies. Several coding standards for source code have emerged to achieve the same high level of safety in software products. Most cover the C programming language, but there are also standards for C++:

MISRA-C++ MISRA stands for Motor Industry Software Reliability Association. They had already published several coding standards for C before they also created MISRA-C++[67] — coding directives for C++[65, 66]. The most important things to know is that they allow RTTI and exceptions in source code but no dynamic memory.

AUTOSAR Guidelines for C++14 While MISRA C++:2008 bases the C++03[45] standard, AUTOSAR provides an update of the MISRA C++ covers the new features of C++ up to version 14[46].

Joint Strike Fighter Coding Standards In 2005, Lockheed Martin released a C++ coding standard[59]. Hence, Lockheed Martin created this standard before MISRA stated its C++ standard.

5.2 Lessons Learned from Software Faults in Space Missions

Even with modern equipment, space missions failed and still keep failing. There are many causes for crashing or losing rockets or satellites. One of them is software failures. When starting with new onboard software, developers should learn from previous failures to avoid running into the same problems and errors. This section gives an overview of several failed space missions that are the results of software failures. Be aware that the list is not complete.

5.2.1 Explosion of the Ariane 501

In 1997, the first flight of Ariane 5 (with the serial number 501) should bring four satellites into space. However, about 40 seconds after the start, the rocket exploded.

The convened inquiry board investigated the issue[28, 53, 60] and found out that a software failure led to the loss of the rocket.

The inertial reference system (SRI), responsible for determining the attitude and movement, was directly taken from Ariane 4. However, the trajectory of Ariane 5 was different from Ariane 4; Ariane 5's initial acceleration is higher than the one of Ariane 4. The different trajectories of Ariane 5 provided higher horizontal velocities than Ariane 4 (about five times). Consequently, the sensors reported greater values than in Ariane 4 before. However, the software was not prepared for such great values and, thus, suffered from an overflow. The overflow led to an interrupt of the SRI. As a consequence, the system switched to the hot redundant SRI. However, the redundant SRI ran the same software and, thus, suffered from the same bug. Hence, the rocket did not have any attitude and position information anymore and initiated self-destruction. In the investigation, the board found that there have not been any tests done with Ariane 5's trajectory.

Lessons learned Always test the software with the consequences of the real environment and the planned maneuvers. In addition, test specifications and test the software's behavior with greater and smaller values than expected.

5.2.2 Crash of the Mars Climate Orbiter

In 1999, the Mars Climate Orbiter traveled to mars and started descending into a stable orbit. During the maneuver, the orbiter disappeared behind the mars 49 seconds earlier than expected. However, it did not come back; it has crashed to the surface.

Also here, a software failure led to this crash. However, the problem was not on board but the ground. The ground software consisted of different modules, which different partners have contributed. Even if there was a Software Interface Specification (SIS) document, one partner did not correctly adhere to the specification. Instead of using SI units, one software module used imperial units. As a consequence, the module miscalculated the trajectory by a factor of 4.45. The result was that the orbiter came too close to Mars and crashed into its surface.

Lessons learned Ground software is safety-critical as well. Develop and implement ground software with the same care as onboard software. Double-check the compliance with the interface specification, especially the physical units.

One could also use features of programming languages in order to keep values with different units incompatible, for example, different `typedefs` or special `classes`. We have created the requirement REQ-SAT-19 from these lessons learned.

5.2.3 Crash of the Mars Polar Lander

The Mars Polar Lander was a mission of NASA for investigating some climate parameters on Mars. However, the landing was not successful; it crashed on Mars' surface[48]. There were sensors at the landing legs, which shall report touching the surface of the planet. However, those sensors also trigger transiently when the legs are deployed. The software considers a touchdown only if the sensor reports the value twice in succession. However, this was still the case when deploying the landing legs. As soon as the software registers a touchdown signal, it disables the descent engine. In the accident, the software did this about 40 meters above the ground, letting the lander crash onto the surface. However, the most significant software issue was that the touchdown sensing was active at a phase where it should be disabled.

Lessons learned The software must adhere to specifications. In this case, the software did not correctly implement software phases, in which the touchdown sensing should be disabled.

6 State of the Art

Indeed, Corfu is not the first framework for onboard software. There already are frameworks for satellite software available, some even under open-source conditions. This chapter gives an overview of the different frameworks.

6.1 NASA's core Flight System (cFS)

With the cFS, NASA provides an open-source framework that can be reused for different space missions [105]. In order to provide decent portability, they organized cFS' architecture in layers, which this section presents.

Figure 6.1 gives an overview of the different layers of cFS. All components are part of one of three software layers: the component layer. Another layer is the execution platform (EP), which is the basis of the overall software. It provides primarily non-functional services, such as timing, pre-emptive scheduling, and thread synchronization. Additionally, the EP already implements pseudo components, which are an integral feature of the software, such as PUS.

The final software layer is the interaction layer (IL). This tier glues instantiated components and the EP together by creating suitable connectors. The authors expect

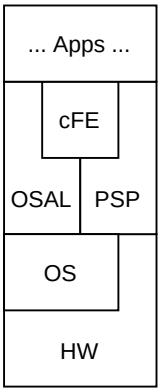


Figure 6.1.: The onboard software structure of the core flight system

to generate code (Ada) for the interaction layer from configuration files; however, this has not been done in their early implementation yet.

Besides that, they started to implement a tool-set that enables graphical configurations of the onboard software. By providing different views for distinct concerns, for example, a data view, a component view, or a hardware view, software architects can concentrate on specific aspects without being confronted with too much information on the screen.

6.1.1 Operating System Abstraction Layer (OSAL)

The OSAL provides a generic interface for accessing and usage of operating system mechanisms and resources. Technically, it could be used independently from the cFE, also for other projects. The interface enables the usage of queues, semaphores, tasks, dynamic loader, timer, network, file systems, and interrupts. To date, there are official implementations for the portable operating system interface (POSIX), the real-time executive for multiprocessor systems (RTEMS)[1], and VxWorks[2].

6.1.2 Platform Support Package (PSP)

Commonly, operating systems are already capable of running on different hardware platforms. Most satellite missions have custom hardware devices, which require additional hardware abstractions. The PSP contains those hardware abstractions. There already exist official implementations for GR-UT699 with VxWorks, MCP750 with VxWorks, PC with Linux, PC with RTEMS, and SP0 with VxWorks.

6.1.3 Core Flight Execution (cFE)

The cFE brings several core services for building up the final spacecraft software. Those services include:

Execution services This is the central part of the framework, which contains the startup and run-time code.

Event services Events represent small messages that the software sends to the ground. Users can use them for reporting debugging information or errors.

[1] https://www.rtems.org/
[2] https://www.windriver.com/products/vxworks/

File services Other than the name suggests, this service provides access to file headers, not to the content of the files.

Software bus This is the primary mechanism for inter-application communication, telecommands from the ground, and telemetry to the ground. We describe the software bus more detailed the following section.

Table services This service manages configuration tables, which applications use.

Time services This service comes with several functions for accessing and converting time values.

6.1.4 Software Bus

The core Flight System's software bus implements the publish/subscribe communication principle. All the routes between the publishers and the subscribers are established dynamically during run-time. Information, such as the routing table and utilization statistics, can be determined by invoking certain API functions.

The software bus uses a single fixed message format at a time. The current implementation only supports one format: CCSDS. The documentation states that it is relatively easy to implement support for other message formats. Subscriptions are created at run-time and connect a message (ID) with a pipe (FIFO). Due to the nature of dynamic subscription creation, it is possible to unsubscribe from messages as well.

Routing tables internally encode subscriptions, the connections between message IDs and the pipes. The software bus implements routing tables using a map with the message IDs as the key and a pointer to the first element of a double-linked list as the value. The list contains all destination pipes for the given message ID.

Every time the software bus receives a message, it retrieves the appropriate pipe list by accessing the corresponding field in the message ID map. The message body is copied and placed into the destination pipe for every entry in the pipes list. The context of the destination application does not execute any receiving code. Instead, the destination applications are responsible for reading the message data from their pipes within their run-time.

Currently, the software bus only supports CCSDS packets for all types of communication, internal and with the ground. Implementing other message formats requires changing the original code of the software bus.

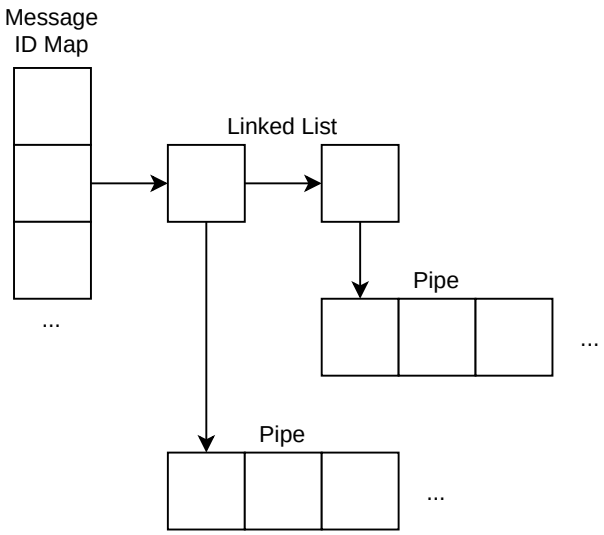


Figure 6.2.: Structure of the software bus of the cFS

Software Bus Spanning Multiple Instances

The pure software bus, implemented within the core Flight Executive, only supports local message delivery because it only directly accesses local memory structures. In order to communicate with other nodes, e.g., with computers connected to a network, there is another application, the Software Bus Network (SBN). It provides transparent communication for the users of the software bus, i.e., the applications do not have to know any information about the communication paths; the SBN takes care of it.

The current implementation requires an IP-Stack to be present on the operating system. A configuration file contains all information about addressing other nodes, which the software loads at startup. The connecting nodes exchange information about which messages the local software subscribes. Whenever there is a change in the subscription status, e.g., a message is freshly subscribed or unsubscribed, the SBN notifies all neighbor nodes. Based on this information, every node is capable of deciding where to forward messages. Additionally, periodic heartbeat packets show that a node is still alive.

6.2 F'

F' – also called F Prime – is a framework for onboard software[10]. The JPL (Jet Propulsion Laboratory) developed F' and made it available under open source

conditions[3]. This software framework resembles Corfu the most. Here, the user also defines the software formally in configuration files. In contrast to Corfu, F' goes for XML as the configuration file format. Although XML offers more elaborated tools and languages for schema validation (e.g., XSD), it is more fluent to write for users. For schema validation, Corfu comes with its own validator.

F' uses the configuration file for code generation. Similar to Corfu, F' generates base classes, which users inherit. The generated base classes define the interface only, i.e., the task interface and the communication ports. On the other side, Corfu also includes thread configurations, which means that even timing information is formally defined in Corfu.

For communication between applications, F' comes with the concept of a port. Every application defines the output and input ports it provides respectively requires. F' provides different types of ports, such as synchronous ports by direct function calls, asynchronous ports by message queues, or guarded ports, which are thread-safe. Differently to Corfu, F's ports use return values, which allow a direct response. In Corfu, developers must report the result explicitly on different topics.

F' is written in C++, and therefore comes with an object-oriented programming interface.

Also similar to Corfu, F' software uses a two-level hierarchy. In the lower level, it defines applications. Those applications consist of input and output ports of different types. In the upper level, topologies connect the applications. Topologies describe how the ports of applications are connected. They are used to compile deployments, e.g., for flashing them on a device.

At Corfu, we name those topologies nodes because they describe the communication topology between applications and check the set of applications for cooperation. Applications do not only influence each other by communication; they run on the same platform and, therefore, have to share limited resources. Such limited resources are devices that applications may and resources in CPU time and memory. In Corfu, we intend to see the whole picture by formally verifying that all applications safely use the resources across all applications and that threads can meet the timings.

F' comes with several reusable software components (applications) that are common for most onboard software. It ranges from commander components over logger and memory components to ground interfaces. Apart from generating source code used in the flight software, F' generates test classes for unit and integration testing.

[3]https://nasa.github.io/fprime/

6.3 Cordet-2

In two steps, Cordet and Cordet-2, an ESA project developed a component-based reference architecture for onboard software[72, 80, 97]. Their idea is the separation of concern into individual components. Containers embed such components, which define the communication interface to other components, more precisely to other containers. The interfaces specify in- and outgoing connections linked by connectors, e.g., a communication middleware channel/topic.

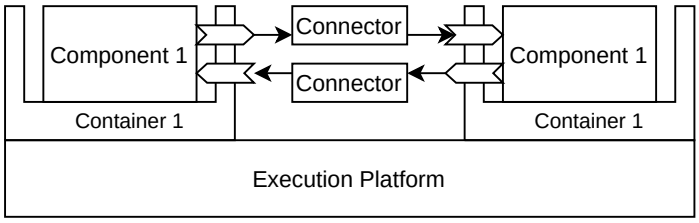


Figure 6.3.: Structure of software following the Cordet architecture (adapted from [72, 97])

6.4 NanoSat Mission Operations Framework (NanoSat MO Framework)

With OPS-SAT, the ESA is working on a cube sat[22] project, which has the aim to test new technologies[33]. One part of the development is the implementation of a new software framework called NanoSat MO Framework. In the name, the MO stands for mission operation, which also depicts its conceptional heritage: it implements the CCSDS Mission Operation (CCSDS MO) standard.

They base their structure also on the concepts of applications like smartphone systems do. Such applications can be stopped, started, and even updated during the run of the onboard software. This dynamic reconfiguration is made possible by the flexible definition of communication interfaces and the discovery mechanism, which loosely couples and enables applications to find each other.

The NanoSat MO Framework uses Java as the programming language. Hence, it needs an entire Java virtual machine to run on the target platform. Therefore, this implementation of the CCSDS MO is not suitable for bare-metal development. In the reference implementation for OPS-SAT, they run Linux as the base operating system which executes the framework.

6.5 OBS framework

The OBS framework from Cechticky et al. follows a generative approach[19]. It seems to be a continuation respective generalization of the AOCS framework because the OBS framework reuses many ideas from the AOCS framework, which is described in detail by Pasetti in [73, 74]. In their paper [20] they describe three technical approaches they use in the OBS framework to accomplish re-usability of software by architecture[75]: feature modeling, object-oriented framework, and aspect-oriented adaptability. Additionally, they describe on their homepage[4] several design patterns, particularly for onboard software. The frameworks further come with already implemented components, such as telecommand and telemetry managers.

Aspect-Oriented Adaptability

In [9], Birrer et al. describe their implementation of aspect-oriented adaptability in the AOCS framework. With their tool named XWeaver[5] they apply aspect weaving, i.e., compiling aspect code into some existing code. The OBS framework also uses XWeaver. It transforms the base C++ code into *srcML*, an XML representation of the abstract syntax tree. The aspect code, to be interwoven, is written in a self-developed language called *AspectX*, which is XML-based. By using rules, it describes how XWeaver modifies the base code in order to import the aspect there.

XWeaver first compiles the AspectX rules into an XSLT program and applies it to the srcML representation of the base code. The result is again an XML-base srcML representation of the modified code. After that, XWeaver transforms the srcML information back into C++ code.

6.6 Ziemke, Kuwahara, Kossev

Those three authors present in [108] a generic onboard software framework they have created. It is an object- and service-oriented framework built on a real-time operating system, with RTEMS being the primary target. They divide software using the framework into four layers: the operating system, the device handler layer, the data pool, and the controller layer. Each of the layers has its own FDIR plugin that, on the one hand, deals with failures, which the software can handle locally, and on

[4] https://www.pnp-software.com/ObsFramework/doc/Home.html
[5] https://www.pnp-software.com/XWeaver/

the other hand, reports failures to the upper software layer. The central part is the data pool; it is one method for data exchange between different applications, e.g., device handlers and controllers. In a transaction-based manner, applications can write into variables of the data pool respectively read from them. All actions onto the data pool are thread-safe.

Device handlers are the interface between device drivers, e.g., sensors, actuators, busses, and the data pool. Commands to hardware devices are saved by controllers in the data pool and executed by the device driver. Receiving data from the device works the other way around; the device handler reads data from the hardware and puts it into the data pool.

However, there are additional communication types between applications, such as messages queues for telecommands, telemetry, and telecommand verification and acknowledgment. Another type of communication is events to indicate mode changes within the software. Finally, there are also signals for FDIR, which leads to preemption of the current process to execute failure handling there.

Users can define the initialization aspects of the software via XML. An XSLT program generates corresponding C++ code. Unfortunately, the authors did not publish follow-up papers.

6.7 Prochazka et al.

In [77], Prochazka et al. present a component-oriented framework for onboard software. They extend SOFA 2, a component definition system, to describe components and their interfaces. Components can be connected when the providing component's interface is a superset of the required interface. Additionally, a component might contain multiple controllers, which bring non-functional aspects to the component.

6.8 Other Related Work

Code feedback for model-driven development has also been used in other environments. For example, Büchner has applied this approach in his Ph.D. thesis for web development[15]. In his work, he calls this approach "introspective model-driven development." Büchner distinguishes between white-box and black-box introspective approaches: "Introspective black-box frameworks realize domain-specific languages

through external model representations, which, however, are integrated at any time with the framework's extension capabilities. In introspective white-box frameworks, models are internally represented by the source code of the base programming language. Through introspection, these models can be extracted and represented and edited at a high level of abstraction."[15] In his work, he created tools integrated into the IDE eclipse for creating introspective white-box frameworks. He evaluates his approach by implementing a web platform. His tools rely on features of the Java programming language, such as annotations. Most embedded satellite software makes use of C or C++, which makes them not directly applicable for the satellite software domain.

Part III

Design of Corfu

7 Methodology

The subject of this work is the theory and implementation of a technical system. Consequently, we apply the Design Science Method[13, 103]. In this methodology, we build a framework for satellite software.

7.1 Development Process

Figure 7.1 shows the development process for Corfu. In the first step, we formulate the goals of the framework (see section 7.2). From there, we have derived several requirements for different aspects (see section 7.3). Based on those requirements, we accomplished two steps in parallel. Following the test-driven development[52] approach, we define unit tests for the intended software features (see section 7.4) before implementing the features in order to fulfill the defined test code. After the tests ran successfully, we also had a manual code review.

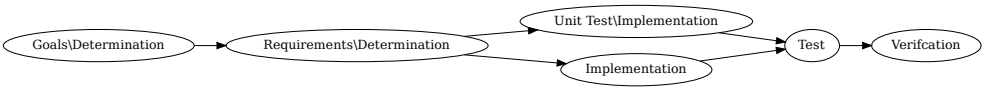


Figure 7.1.: Development process for creating Corfu

7.2 Goals

With our framework, we pursue different goals, which we describe in this section.

Improve Reliability For safety-critical applications, reliability is essential for both hardware and software. Reliability defines the probability that failures occur during operation. IEEE states in [43] that the software reliability depends on the development and maintenance process of the software. Therefore, our framework has to come with a robust guide for developing reliable satellite software.

Facilitate the Development Process Following development approaches like model-driven development (MDD) already provides a clear development path. Having only one way of how to introduce new features and changes avoids dangerous quick fixes. Providing appropriate tools and a high level of automation takes much work off the developers' shoulders. Indeed, such automatic tools have to be well validated and verified because they produce safety-critical software.

Reduce Development Time Having fewer lines of code to write manually often comes with less development time for the software. Corfu shall aim to generate as much boilerplate code as possible to disburden manual coding from developers. Instead of bothering with boilerplate code, they can concentrate on developing the essential features of the onboard satellite.

Another way to reduce development time is to make it easy to understand the software structure. Corfu's model shall describe the software structure concisely and lucidly (REQ-MDD-03). Representing the model additionally in a graphical way helps developers find their way around the model configuration (REQ-MDD-05). Having a graphical overview also reduces training time for new developers.

Improve Ground Segment Satellite Communication The ground software is a companion of the onboard software. Both have to interact with each other. Therefore, another goal shall be the good integration between the space and the ground software. Hence, the framework shall come with both onboard and ground software support (REQ-GND-03). However, the framework should be flexible enough to integrate into custom ground software easily (REQ-GND-04).

Extend the Model-Based Development Approach Just following the MDD approach in one direction does not exploit the full potential. We show that introducing feedback improves performance and enables the implementation of new software features into OBSW.

7.3 Requirements

We have collected more than 30 requirements in those different categories:

- Satellite Software

- Safety-Critical Software
- Applying Model-Driven Development
- Model Feedback
- Embedded Software

Appendix B list all of those requirements. These requirements are the foundation for developing Corfu.

7.4 Testing

These unit tests cover the source code of Corfu itself, not of the user code. We use google test[1] as the driver for Corfu's (internal) unit tests. Each application and library of Corfu comes with its tests that aim to cover the highest possible coverage. For the user code, Corfu comes with its own testing framework as chapter 14 describes.

[1] https://github.com/google/googletest

8 Basic Concepts and Design of Corfu

This chapter describes the basic concept of Corfu. It starts with the use cases, which we have identified from the goals and requirements. After that, the chapter gives an overview of the development process and its tools. In the end, it describes the structure of the onboard software and different software concepts, such as telecommand handling and standard telemetry.

8.1 Use Cases

We divide the users into three different groups: engineers, developers, and operators — see the use case diagram in figure 8.1. **Engineers** design the software structures. In model-driven development, their main work is to define one or more models of the software manually. Defining models includes all model aspects, such as defining apps, nodes, and communication structures. They usually design the model based on given requirements. Engineers can directly check several model aspects by applying verification tools, such as a consistency check and scheduling analysis. The consistency check verifies whether required parameters are present in the model, whether they contain correct types, and whether all referenced objects exist.

Developers write source code. The first step generates code from the model(s), which engineers have defined. The framework and the generated code provide attachment points, such as virtual functions, where developers can implement their code. Finally, developers can compile the onboard software and test their code.

Up to this point, all actions take place at compile time. **Operators**, on the other hand, work at run-time, i.e., when the satellite is launched and in orbit. They monitor and control the satellite with telemetry and telecommands. Telecommands that they send to the satellite might generate extended telemetry or report anomalies. The onboard system executes threads, which also might generate extended telemetry or report anomalies. There is at least one thread that also periodically generates standard telemetry.

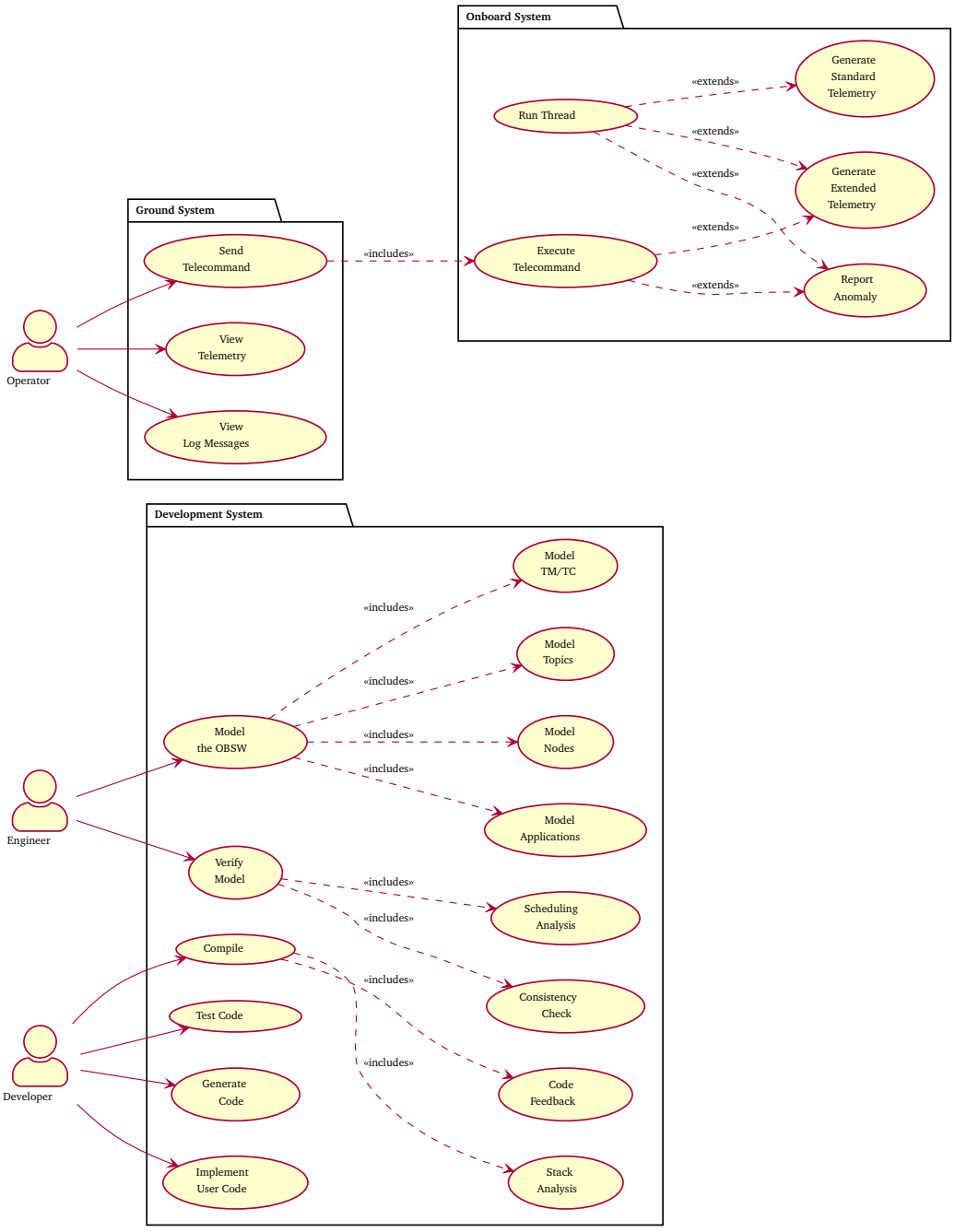


Figure 8.1.: Use case diagram of satellite systems

8.2 Development Process

Corfu provides an iterative development approach. Iterative means that the first version of the model does not have to be the final one. It is always possible to refine the model and re-generate code. Figure 8.2 shows an activity diagram of the development process in a project's lifecycle. The figure covers the iterative character of software development. There are two loops in the development process. The outer one describes the iterative refinement of the process. Whenever engineers modify the model, they re-create the generated code, which is the input for the developers. The inner loop describes the iterative development of the user code.

Figure 8.3 shows the technical process of Corfu. It displays how documents (artifacts) and tools work together. The arrows show the information flow between documents and tools. We have graphically highlighted two parts in the figure. The blue part covers the classical model-driven development. Outgoing from a model, a generator creates code, which is finally compiled together with the user code and additional libraries. The orange part is the feedback, which we introduce in this work. It adds further information to the existing model, which verification and code generation also use.

When executing the compilation process for the first time, the orange step is not applied because no user code exists at this moment. In this step, the **code analyzer** extracts information from the manually written source code of the developers (**see chapter 12**). The extracted information represents the **extracted model**, which the code analyzer stores into a SQLite database (**see section 12.1**). The extracted model is optional for the generator and model verification. However, those tools always require the engineering model. Therefore, at least a first draft of the engineering model is mandatory when starting the development.

The **generator** takes information from both models, the formal and extracted one, and generates source code files from source code templates. See **section 11.5** for a detailed explanation of the code generation process. Finally, the compilation and linking step combine the generated, the user source code, and **Corfu's libraries** (**see chapter 10**) to create the binary file(s).

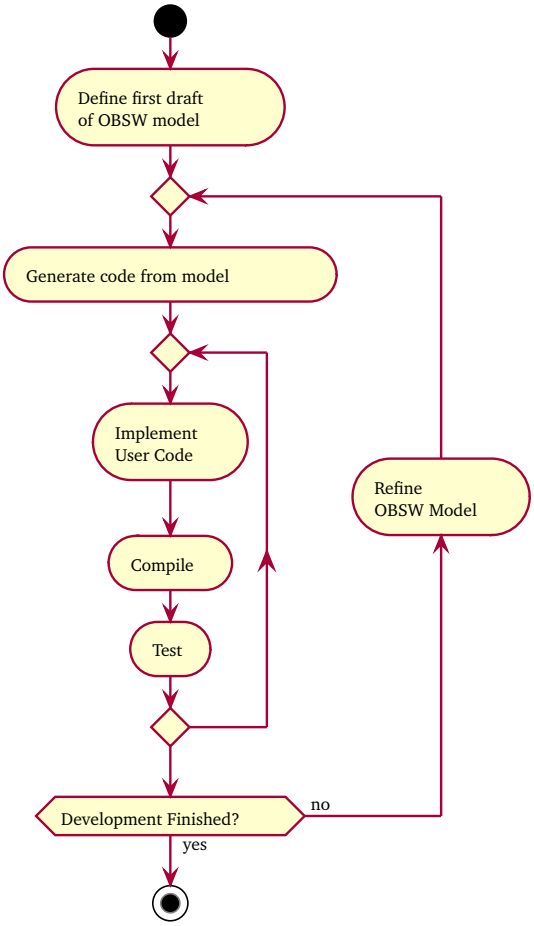


Figure 8.2.: Activity diagram of the development process

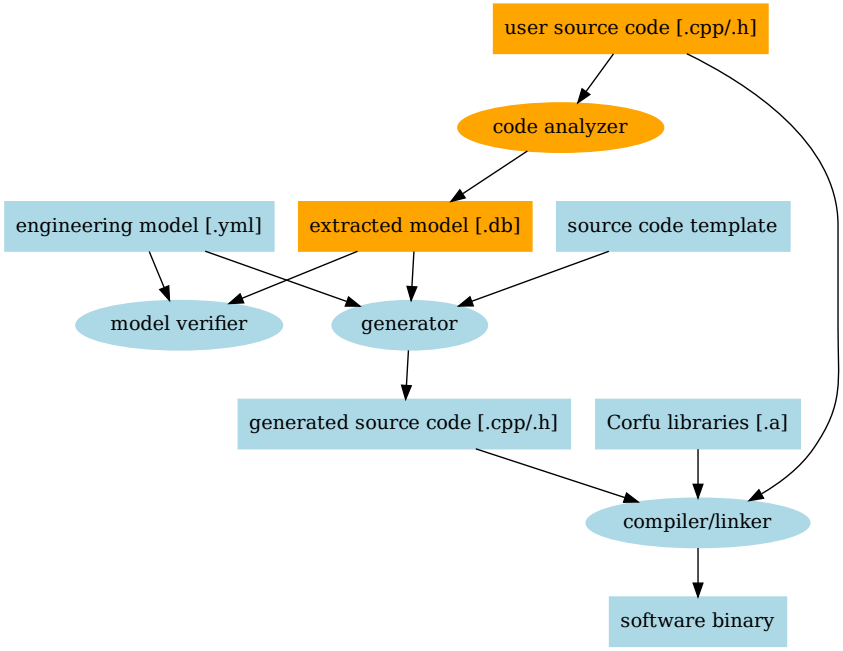


Figure 8.3.: Corfu's compilation process

8.3 Static Structure of Onboard Software

Most frameworks promote the reusability of software parts; so does Corfu. It allows partitioning the software into separate units with independent features. A communication interface allows the applications to communicate with each other.

In Corfu, we have two different software components: applications and shared resources. Applications represent active parts of the onboard software. They may contain threads, which actively drive the execution of code. In addition, they take part in onboard and ground communication. That means they can subscribe to topics and provide telecommands to the ground software. Figure 8.4 shows a schema of app components. Both upper topics, telecommand, and telemetry are common for all applications. The lower blue boxes stand for application-specific topic usages. The green boxes contain user code, and the orange boxes contain data (e.g., variables).

Shared resources are passive elements. They do not have any threads or provide telecommands. Instead, they are just there to be accessed from applications. `ThreadSafeDatas` wraps shared resource, which ensures thread-safe access (see 17.1.1 for more details).

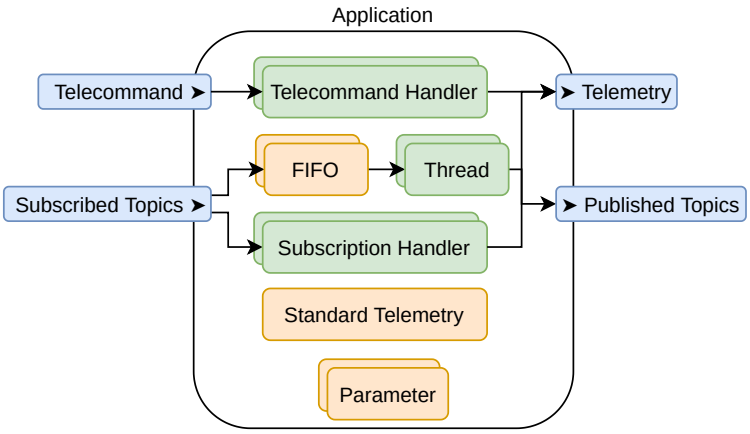


Figure 8.4.: Configurable application elements in the model

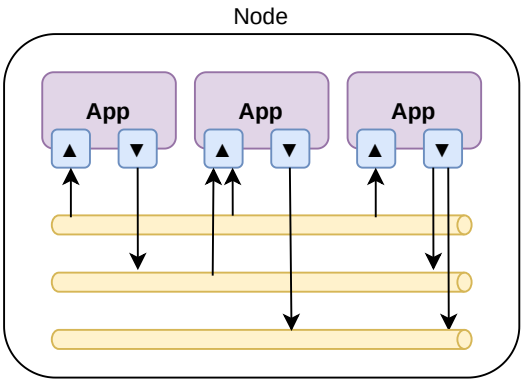


Figure 8.5.: Configurable node elements in the model

Applications and shared resources stand for their own. In order to instantiate a full executable software, we have to select some and connect them, which nodes do. Nodes represent software images running on a computer. For the onboard communication, we use the publish/subscriber communication middleware provided by Rodos. Nodes connect apps via topics (communication channels) with each other. Figure 8.5 shows a schema of node components. It depicts topics as communication channels, which applications subscribe (arrow towards the application) or publish (arrow towards the topic).

Corfu divides the overall source code of onboard software into three different layers: generic, generated, and user code, which figure 8.6 shows. The diagram shows only a simple application that only implements two telecommands and standard telemetry. It does not subscribe to any topic or uses additional telemetry.

The abstract class `corfu::App` is common for all applications. This class contains generic telecommand handling and provides helper functions for sending telemetry. See section 8.4 for an overview of those concepts.

The next level of application classes is a generated abstract class (`generated::MyApp` in the figure 8.6). The generator creates it for each application in the software configuration. That means it contains the specific code for handling telecommands, telemetry, topics, and threads. The generated class also defines abstract handler methods, which the developers have to implement, e.g., a telecommand handling method.

Finally, developers write their user class (`MyApp` in the figure). This class only has to inherit from the generated class. Developers have to implement the abstract handler methods with the desired behavior. The generated class preprocesses all messages that arrive at the application and calls the appropriate handler methods. See section 8.4.1 for more details about how applications handle telecommands.

Figure 8.7 shows the hierarchy of generated nodes. For brevity, the diagram only shows those classes and methods that are part of the general node hierarchy. Other classes, e.g., those that are part of the telemetry handling, are shown in the section 8.4.

Figure 8.7 shows the hierarchy of node classes. Similar to the classes for applications, nodes also use inheritance. However, for nodes, we have only two classes: the base class and the generated class. Creating a user-written class is unnecessary because the generated class already contains everything a node needs. The generated class instantiates and configures all the defined applications according to the software configuration. The class `corfu::Node` is the generic base class that all nodes use. It contains code that is independent of the software configuration.

8.4 Concepts

Apart from the static definition of nodes and applications, Corfu comes with concepts of dynamic processing information. This section gives an overview of how telecommands and the standard telemetry are processed and how the software watchdog works.

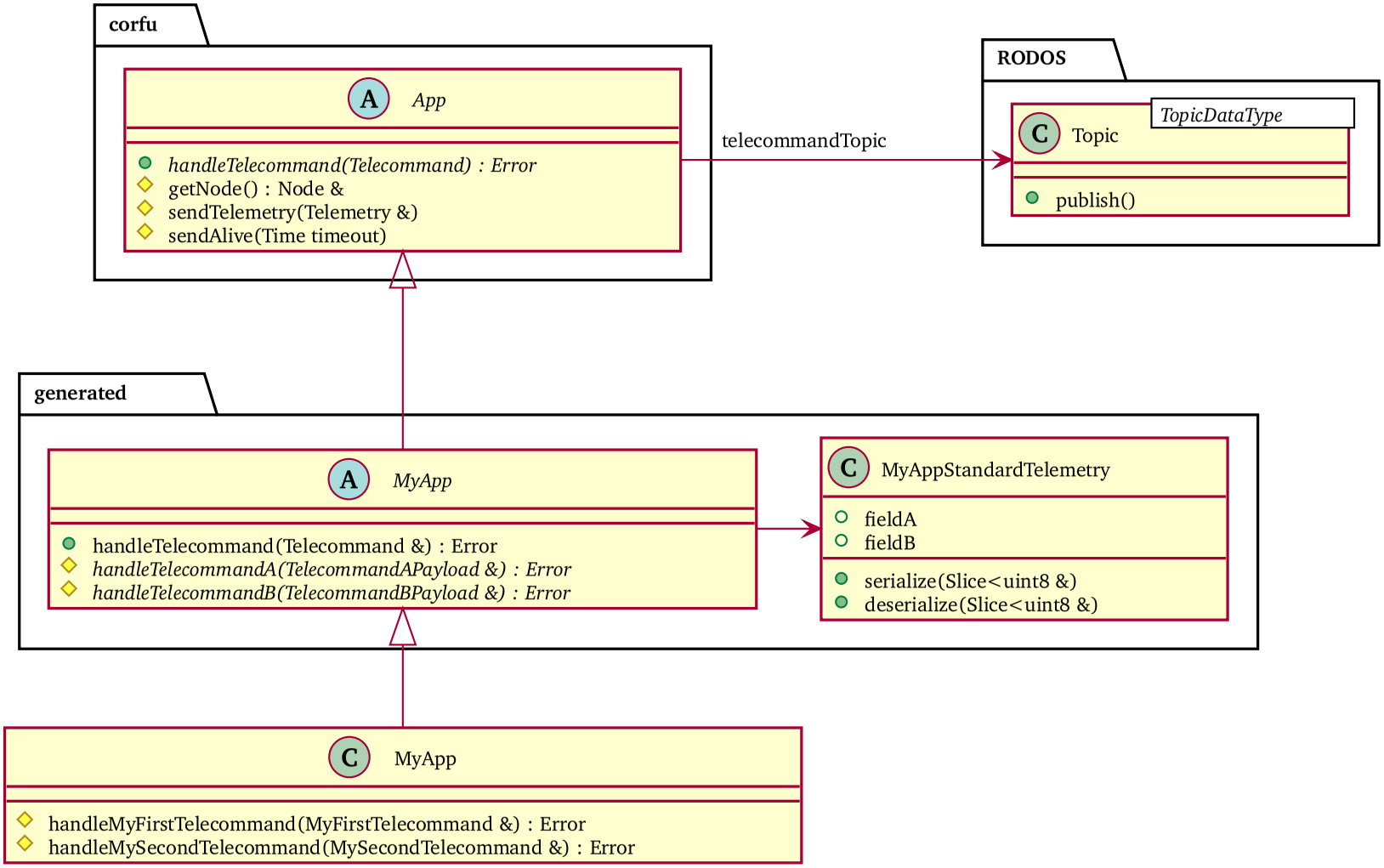


Figure 8.6.: The hierarchy of application classes in the Onboard Software

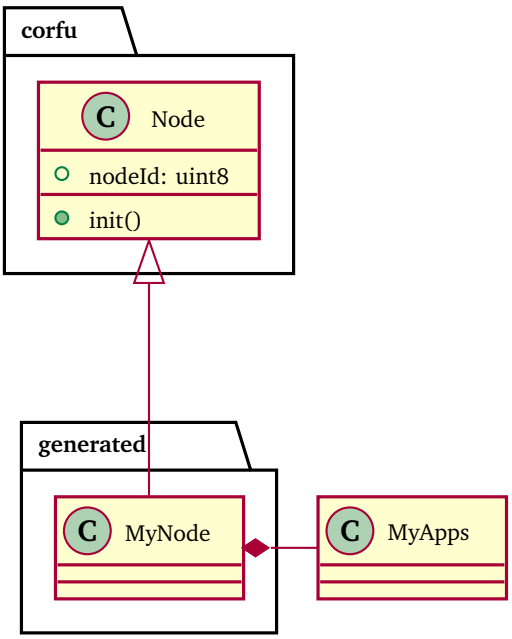


Figure 8.7.: The hierarchy of node classes in the onboard software

8.4.1 Telecommand Handling

On board, topics distribute telecommands. How they arrive at the space segment is mission-specific. For example, a modem driver (application) receives telecommands from the ground, decodes them, and publishes them to the `telecommandTopic`, which applications subscribe to. The data type of this topic is the generic telecommand class `corfu::Telecommand`. Apart from a checksum and the parameters, the telecommand class contains information utilized for routing. The `nodeId` identifies the node to be addressed; the same goes for the `appId`. Finally, the `commandId` identifies the type of telecommand for the application.

Besides the application class, the generator also creates telecommand classes for each type of telecommand. The generated telecommand classes contain not only the parameter values but also methods for serialization and deserialization.

The sequence diagram in figure 8.8 shows how applications distribute the messages internally. As soon as the application receives a telecommand via the `telecommandTopic` , the `corfu::App` class forwards the message to the `handleTelecommand` method. Only the generated application classes implement this handling method. First, the method checks whether the telecommand addresses the application. If that is the case, it deserializes the telecommand parameters into the telecommand-specific object. Then, the method passes the deserialized object to a telecommand-specific han-

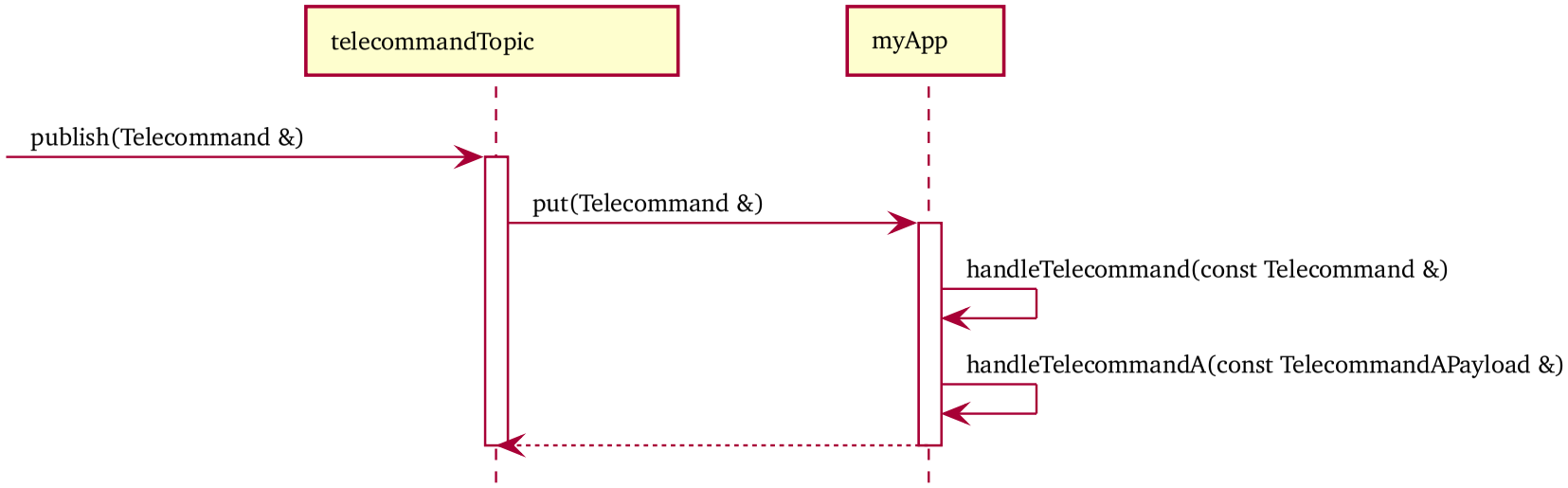


Figure 8.8.: The sequence diagram of telecommand distribution in applications.

dling method. Such telecommand-specific methods are abstract in the generated application class but implemented in the user-written application class.

8.4.2 Standard Telemetry

The idea behind standard telemetry is to provide brief information about the satellite's state to the ground operators. Each application contributes its own structure with a few fields that indicate the state of the application. The information of the applications should not go into much detail. If the operation crew needs extended information, they shall request extra telemetry, which applications provide. Finally, the satellite periodically sends the resulting telemetry structure with the information of all applications down.

Figure 8.9 shows all classes involved in collecting information for standard telemetry. The generator creates a specific struct equipped with serialize and deserialize methods for each application that contributes data to standard telemetry. The generated application class already contains an instance of the telemetry structure as a member variable. The applications directly fill this structure's data. Applications like the housekeeper invoke `serializeStandardTelemetry` to collect the node-specific standard telemetry data into a slice/array. The node traverses all applications and calls their serializing methods. The node calls the `updateStandardTelmetry` method right before serializing the data to ensure that the data in the member instances of the telemetry structure is current. Overriding `updateStandardTelmetry` gives applications the chance to update the structure with current data if they are not already up to date.

8.4.3 Periodic Thread

Listing 8.1 shows how we can create periodic threads in plain Rodos. We have three different parts of user code and, most notably, a time loop macro, which controls suspending and resuming the threads at the given times. The first part is initialization before the scheduler runs in the `init` method. Here, we can only apply initialization, which relies not on the scheduler, e.g., no suspension. The `run` method is the entry point for the thread. As soon as the scheduler is ready (and the thread has the highest priority among all other active threads), it calls the `run` method. Therefore, the code has access to all features here. Hence, initialization which requires scheduling features, such as suspension, can be implemented here.

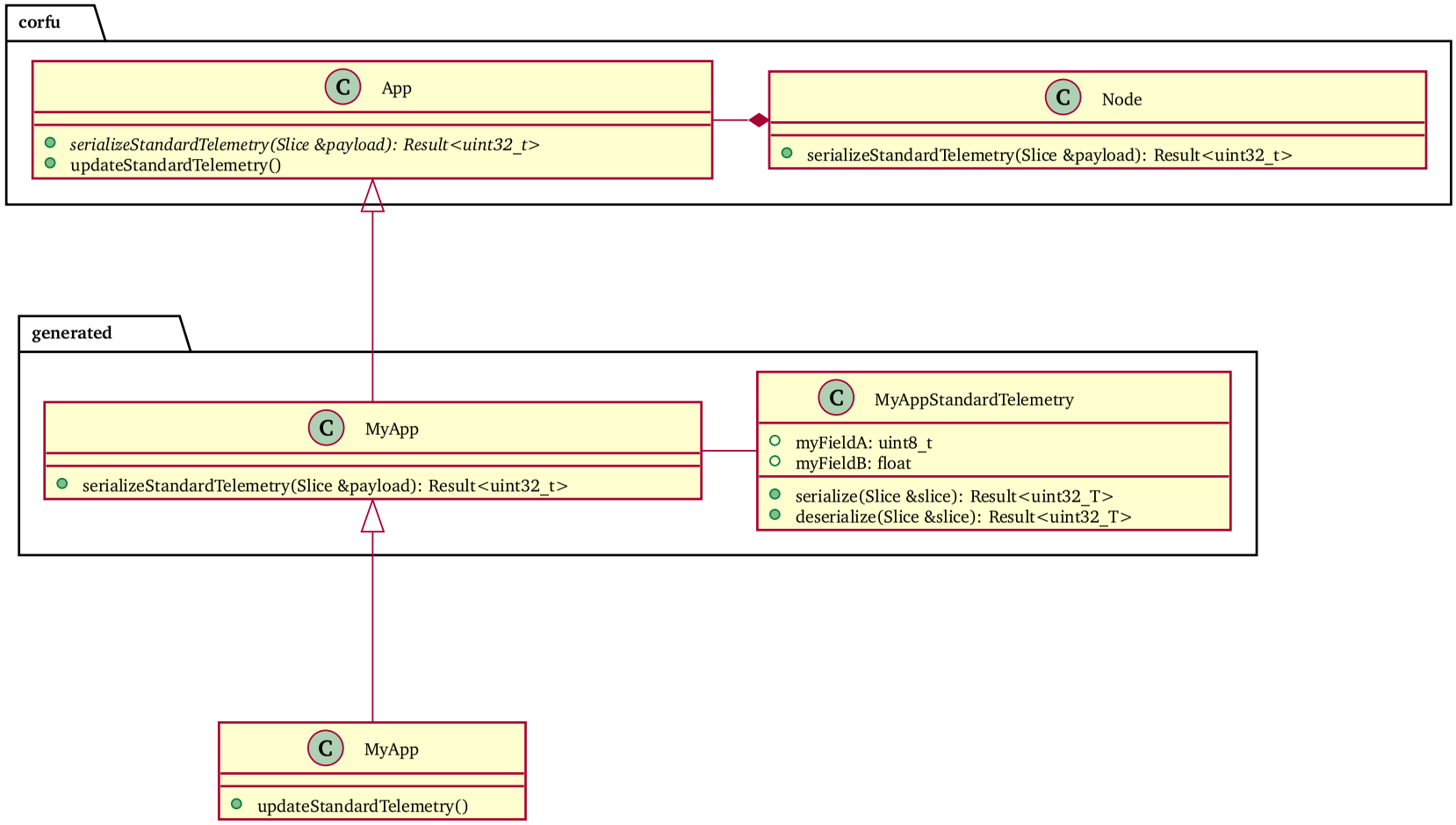


Figure 8.9.: The class diagram of collecting standard telemetry.

The `run` method also contains the time loop, which runs infinitely. It executes its body according to the timing parameters.

```
constexpr int64_t FIRST_RUN = 1_s;
constexpr int64_t PERIOD    = 500_ms;

class MyThread : public RODOS::StaticThread<> {
  public:
  void init() override {
    // initialization before the scheduler runs
  }

  void run() override {
    // initialization while scheduler runs
    TIME_LOOP(FIRST_RUN, PERIOD) {
      // periodic code
    }
  }
}
```

Listing 8.1: Classical approach to implement periodic threads in rodos

In order to keep this structure consistent for all periodic threads in the onboard software, Corfu introduces its own class for periodic threads, `corfu::PeriodicThread`. It inherits from `RODOS::StaticThread` and directly passes the template parameter for the stack size, see also figure 8.10. The part with the time loop is implemented in `corfu::PeriodicThread`'s `run` method. The loop's body does nothing else than invoking the virtual method `runIteration`, which the user class implements. Both initialization methods are also called by `corfu::PeriodicThread` (see figure 8.11). Their name make clear, when they are called. `unscheduledInitialization` is invoked by `corfu::PeriodicThread::init` and `unscheduledInitialization` is called in `corfu::PeriodicThread::run` before the time loop is entered.

For each thread defined in the model, the generated app class contains virtual methods for the user code. Only the `runIteration` method is purely virtual, i.e., mandatory for the user to implement. Both initialization methods have a default implementation, i.e., they are optional for the user to implement. Each application instantiates an appropriate subclass of `corfu::PeriodicThread`, which does nothing else than invoking the associated user methods.

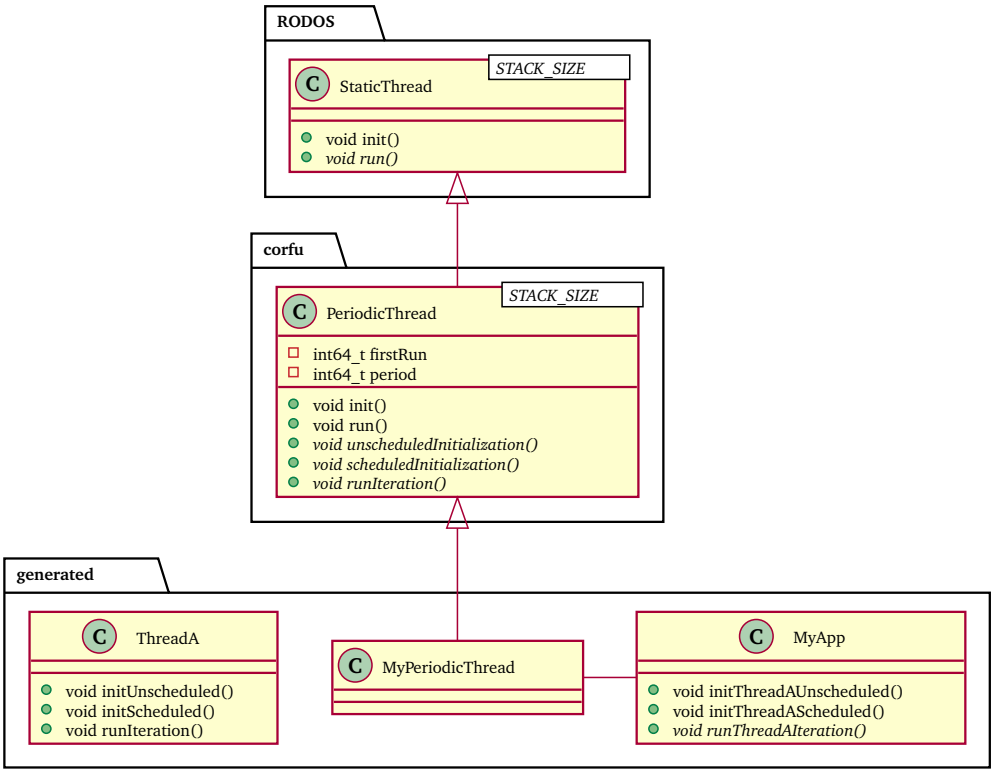


Figure 8.10.: Corfu's approach to implement periodic threads (class diagram)

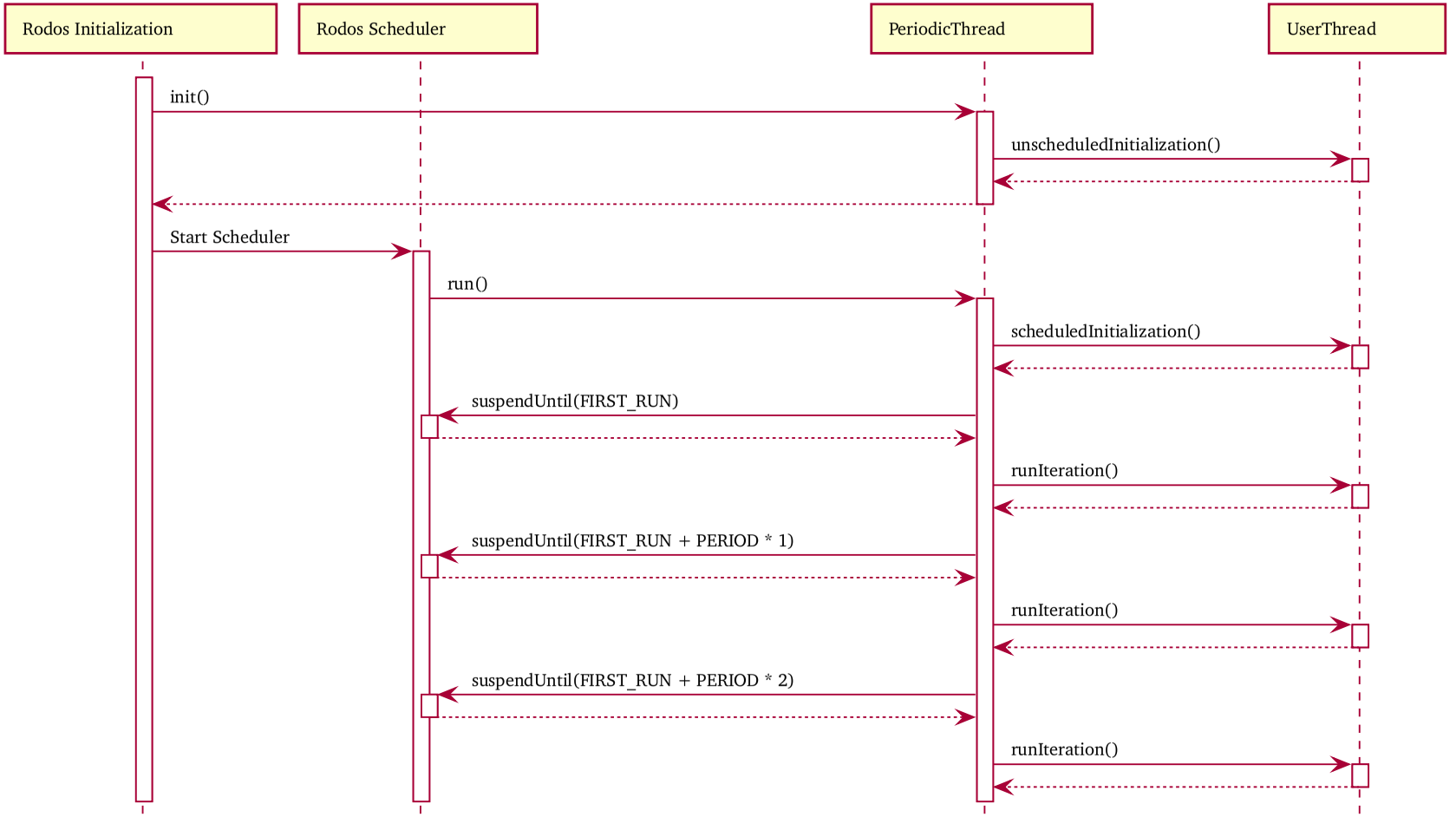


Figure 8.11.: Corfu's approach to implement periodic threads (sequence diagram)

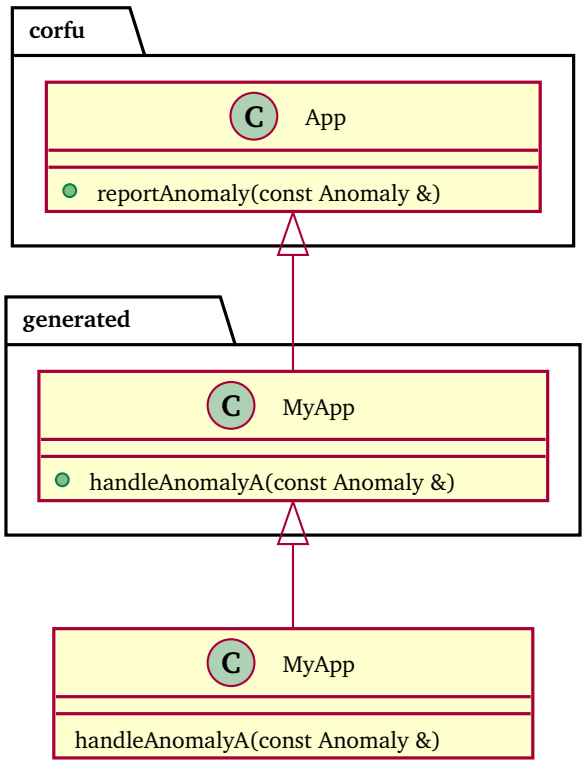


Figure 8.12.: Example class diagram of local anomaly handling

8.4.4 Local Anomaly Handling

Whenever something unexpected happens, applications report anomalies. The generic class that all applications inherit provides methods for the user for reporting anomalies. Applications publish every anomaly being reported to a topic, which will be distributed on the node (e.g. for collecting them in an application). However, as requirement REQ-SAT-08 states: anomalies shall be handled in the lowest possible level. That means, anomalies shall be handled within the application if possible. In order to unify in-application anomaly handling, Corfu generates handle functions for all the anomalies that applications reference in their configuration.

Figure 8.12 shows an example of an application (MyApp) that reports one type of anomaly (MyAnomaly). This class diagramm focuses on the anomaly handling and, therefore, omits all other methods and member variables. The user code invokes `corfu::App::reportAnomaly` as soon as the software detects an anomaly. Corfu's generic app class first calls the appropriate virtual handle function, before it publishes the anomaly to the topic — as the sequence diagram in figure 8.13 shows.

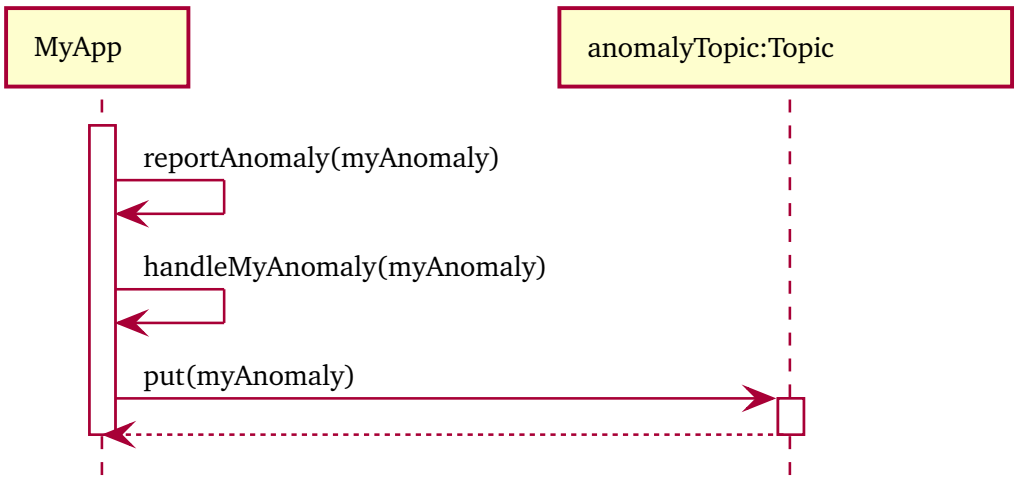


Figure 8.13.: Example sequence diagram of local anomaly handling

8.4.5 Event System

Satellite missions are limited in resources, including the bandwidth of the radio connection between ground and space. On the other side, it is desirable to have a verbose logging system. However, it is undesirable to transmit the same strings repeatedly. To address this issue, we have implemented an event system that uses information from the extended model to reduce the required capacity for transmitting event messages.

Indeed, it would be possible to pre-define all messages in the engineering model. However, our experience shows that engineers seldom know all event messages in advance. Often, the events depend on the actual implementation, e.g., on control structures and defined variables.

Onboard Code

In the source code, developers report event messages by calling a macro in C++. Listing 8.2 shows two example event messages in the onboard software.

```
1 EVENT0(INFO, " Message");
2 EVENT(INFO, "Message", varA, varB);
```

Listing 8.2: Example events being reported in the source code

For each event, developers must pass a string (the message text) and zero or more parameters. To provide an easy programming interface with a flexible number and types of parameters, we have created two macros, which listing 8.3 shows.

```
#define EVENT0(severity, msg) \
  sendEvent((severity), appId, RODOS::hash(msg))

#define EVENT(severity, msg, ...) \
  sendEvent((severity), appId, RODOS::hash(msg), __VA_ARGS__)
```

Listing 8.3: Event macros

The first macro, `EVENT0`, sends events without parameters; the second macro, `EVENT`, takes variadic parameters (but at least one). All events are triggered within the code of applications. Therefore an application ID is available when calling `EVENT`, which is bound to the event. The `hash` function converts the passed string into an integer hash value. The crucial point here is that we defined `hash` function as `constexpr`. Defining it as `constexpr` gives the compiler the possibility to convert the string into the hash value already at compile-time. Therefore, we can transfer the small hash value via radio instead of the big event message. In addition, the string does not end up in the binary file.

In order to transmit the event data, the software has to serialize all parameters. As the parameter values change at the run-time, we cannot move this part to compile-time. However, the number and types of parameters are fixed at compile-time. We use this fact to let the compiler generate the serializing code by providing template functions, which figure 8.4 shows.

```
template <typename Arg>
uint32_t serialize(uint8_t *buf, Arg &arg) {
  return BasicSerializers::serialize(arg, buf);
}

template <typename Arg, typename... Args>
uint32_t serialize(uint8_t *buf, Arg &arg, Args... args) {
  uint32_t bytes = serialize(buf, arg);
  bytes += serialize(buf + bytes, args...);
  return bytes;
}

template <typename... Args>
void sendEvent(Severity severity, uint8_t appId, uint16_t
    hash, Args... args) {
  Event event{ severity, appId, hash, NOW() };
```

```
16    serialize(event.parameters, args...);
17    eventTopic.publish(event);
18  }
19
20  void sendEvent(Severity severity, uint8_t appId, uint16_t
        hash) {
21    Event event{ severity, appId, hash, NOW() };
22    eventTopic.publish(event);
23  }
```

Listing 8.4: Template functions for serializing event messages with their parameters

By applying template metaprogramming[1], we generate a series of `serialize` functions that serialize the parameters one by one. From the viewpoint of the template function, this looks like recursive calls, but the generated code does not call itself recursively. Each `serialize` function cuts one parameter from the parameter list and serializes it into the buffer. The function then passes the rest of the parameters to another `serialize` function until there is no parameter left. For each combination of parameters, the compiler generates a `serialize` function and all subsequent functions, with the first parameter removed in each case.

Whenever a `serialize` function serializes a parameter, it returns the number of bytes that the serialized parameters occupy in the buffer. This value includes the number of bytes for the serialized parameter and the return value of calling the subsequent `serialize` function. Finally, the `sendEvent` function stores the hash value of the message into a `Event` struct and publishes it to the `eventTopic`.

Dannemann described in his dissertation[26] a monitoring framework, which also managed to reduce the binary size and transmission bandwidth. However, he used a preprocessor in his approach, which replaced the messages with alternative C++ code. The alternative code does not contain the string message and instead of an ID. Replacing the string message with an ID is similar to our approach; however, we do not need an extra preprocessor. Thanks to the `constexpr` keyword, which has been introduced to C++, we do not need to change the user's source code at compile time. Therefore, the user code is compiled as it is, just following the C++ standard. Our approach does not require further (special) processing. Not having extra processing tools avoids introducing new potential bugs and problems which are not foreseen.

8.5 Reporting Programming Errors at Compile-Time

There are different ways to detect programming errors already at compile-time. Having error reports at compile-time is excellent for reporting faults before they become failures at run-time.

One approach for avoiding faults is to prefer references over pointers. The code *must* initializes references with a value; otherwise, the compiler raises an error. Hence, there is no null pointer that the code has to test. However, developers can not reassign references with different addresses, making them not a general replacement for pointers.

Another aspect, which the power of ten (see 5.1.1) mentions, is to consider compiler warnings as errors. Consequently, developers shall fix every warning that the compiler reports. Most compilers provide a flag that aborts the compilation when a warning occurs; in GCC and Clang, this is *-Werror*.

9 The Engineering Model of Onboard Software

The engineering model is the starting point for the model-driven approach of Corfu. It lets engineers define structural and behavioral aspects of the onboard software. This chapter describes the integral elements of the engineering model.

9.1 Structural Part of the Engineering Model

Corfu allows developers to define the onboard software's structure in a engineering model. Figure 9.1 presents the structural part of the engineering model as an entity relationship model. The grey Project node is the top-level entity, which represents the onboard software project. It contains a list of nodes that are part of the project. The yellow nodes belong to the node hierarchy level. Nodes have a list of AppInstances, which reference an application and zero or more AppCompileTimeParameters, whose values the AppInstance may override. All the blue entities define the different aspects of apps. In addition to the actual App entity, there are entities containing information about run-time parameters, threads, shared resources, telecommands, and telemetry. The green entities represent complex data structures used by applications topics, telecommands, and telemetry. In the following, we describe the entities in more detail.

9.1.1 App

Application entities are quite simple; they have only two direct parameters:

ID An integer number that identifies the application in a node. Indeed, this value must be unique among all the other applications (the config parser checks this fact).

Name Each application has a name; represented as a string. The name is used for debugging purposes and in the ground software. In addition, it is also the basis for the generated class names.

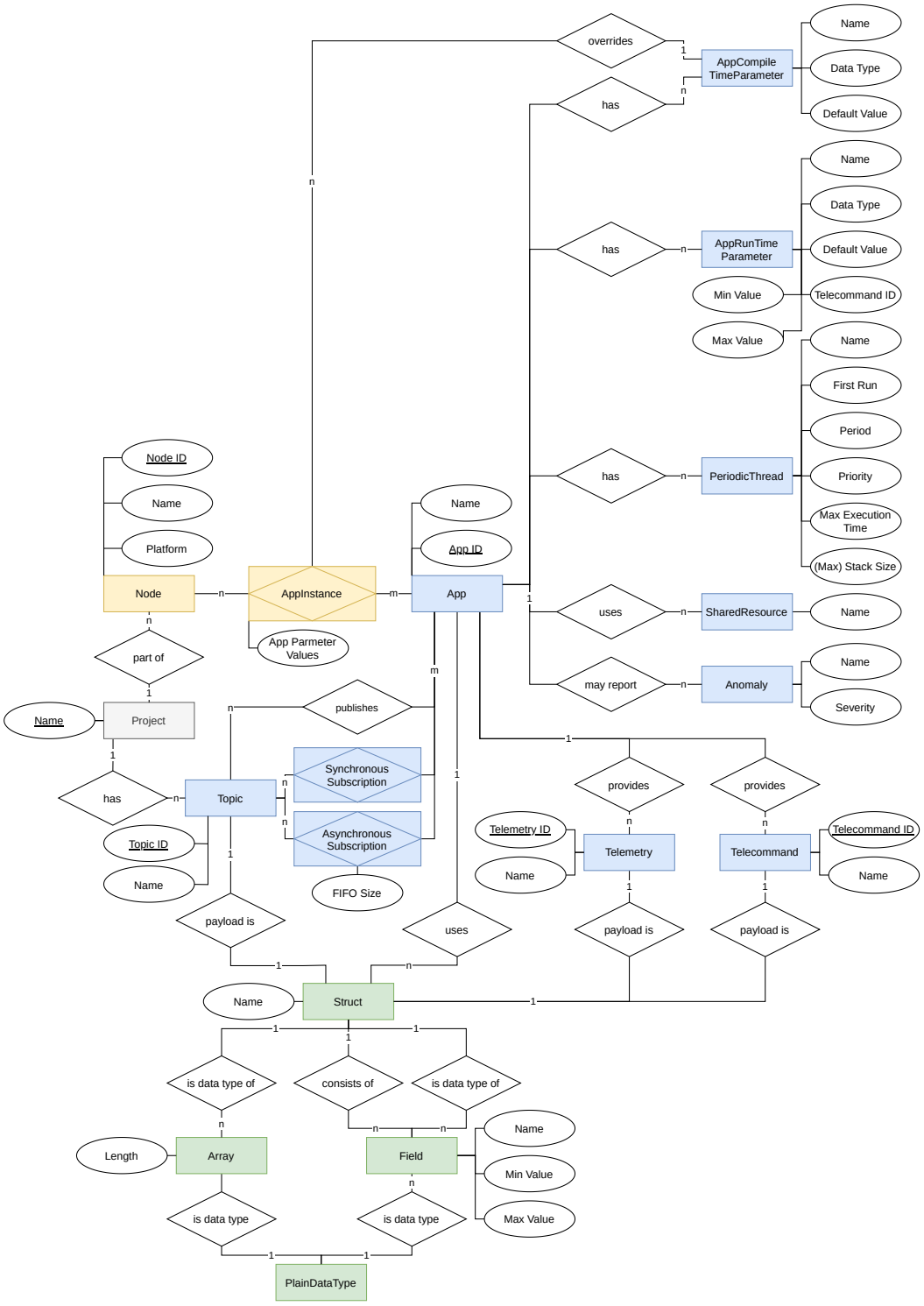


Figure 9.1.: The entity relationship diagram of the structural part of the onboard software in Corfu

Apart from those two direct parameters, Applications reference a lot of other entities as well: SharedResources, SynchronousSubscriptions and AsynchronousSubscriptions, Publications, AppCompileTimeParameters, AppRunTimeParameters, PeriodicThreads, Anomalies, Telecommands, and Telemetry.

9.1.2 Node

Nodes have three direct parameters:

ID An integer number that identifies the node in a satellite. Indeed, this value has to be unique among all the other nodes (the config parser checks this fact).

Name Each node has a name; represented as a string. The name is used for debugging purposes and in the ground software. In addition, it is also the basis for the generated class names.

Platform Each node runs on different hardware, which might require different toolchains to build the software. To use the correct building environment when compiling a node, engineers can specify the platform for each node.

In addition to those direct parameters, nodes reference applications. However, they do not reference them directly; instead, they reference the entity of AppInstance, which in turn references applications.

9.1.3 AppInstance

An AppInstance represents the instantiation of an application on a node. Through AppCompileTimeParameters, it is possible to customize applications on the nodes. Each application instance can define custom values for the AppCompileTimeParameters. Hence, AppInstance contains fields for overriding the default values of AppCompileTimeParameters.

9.1.4 AppCompileTimeParameter

Developers shall design applications to be reusable. However, platforms and satellite requirements differ. Therefore, applications might behave differently on different satellites or even on different nodes on the same satellite. Corfu makes it possible to define parameters whose values are fixed at compile time to make applications

configurable. For example, engineers can use those parameters to specify the length of arrays, which is not possible with dynamic variables that change their value at run-time. AppCompileTimeParameters come with the following parameters in the model.

Name The parameter name identifies the parameter, represented as a string. AppInstance uses it for overriding the parameter's default value and for access in the source code.

Data Type The Data Type defines the basic C++ data type of the parameter used in the generated code.

Default Value Engineers can override the value of parameters in the model. However, if AppInstances do not explicitly override the value, the default value is used, which is defined here.

9.1.5 AppRunTimeParameter

The values of AppCompileTimeParameters are fixed at the compile-time and, therefore, cannot be modified at run-time. However, it can be desirable to reconfigure applications in orbit, which AppRunTimeParameters cover. In the model, engineers can define run-time parameters similar to compile-time parameters. In contrast to compile-time parameters, the code generator creates non-const variables from run-time parameters, which the operations crew can change via telecommands. The AppRunTimeParameter comes with similar parameters in the model as the AppCompileTimeParameter.

Name The parameter name identifies the parameter; represented as a string. AppInstance uses it for overriding the parameter's default value and for access in the source code.

Data Type The Data Type defines the basic C++ data type of the parameter used in the generated code.

Initial Value This is the initial value when the software starts.

Min Value This is the minimum value of the valid range. If this parameter is set, the onboard software automatically checks the value before writing it.

Min Value This is the maximum value of the valid range. If this parameter is set, the onboard software automatically checks the value before writing it.

Telecommand ID If this parameter contains a value, Corfu generates a telecommand for setting this parameter. The auto-generated code does not provide any checks. If users desire value checks, they have to implement appropriate telecommands manually.

9.1.6 PeriodicThread

Threads provide active execution paths, which are activated and executed by the operating system's scheduler. Each thread belongs to an application. There are two types of threads: PeriodicThreads and AperiodicThreads. PeriodicThreads execute their code periodically, at fixed intervals. They come with several parameters in the model.

Name Gives the thread a name. It is not only used for debugging purposes and in the ground software. The name is also the basis for the generated class and variable names.

FirstRun This is a time value that defines the first time point the scheduler executes the thread's code for the first time. The time value is relative to the boot time of the software.

Period This is also a time value. It defines the interval between two code executions.

Priority Real-time operating systems like Rodos often come with a priority-based scheduler; this is also the case for Rodos. The priority value of the thread decides which threads the scheduler executes first. Engineers can define the priority of threads here. The generated code directly passes the priority value to the operating system (Rodos).

Maximum Execution Time This time value defines the maximum execution time the thread might take. Corfu uses this value mainly for the scheduling analysis.

Maximum Stack Size Each thread has stack memory used for the code execution. The required stack size depends on the code that threads execute. Therefore, Corfu gives engineers the possibility to define the stack size for the thread.

9.1.7 Anomaly

The world is not perfect. And so will never a technical system be perfect, including satellites. We must always expect faults. That means we have to detect, isolate, and recover from faults. The onboard software should handle faults as locally as possible. In addition, the software should report faults externally so that the system and the operations crew can respond to faults. Therefore, Corfu enables defining anomalies in the engineering model. Anomalies contain three parameters in the model.

ID An integer number that identifies the anomaly in an application. Indeed, this value must be unique among all the other anomalies in the application (the config parser checks this fact).

Name A descriptive name of the anomaly. The generator creates constexpr variables in the code with this name, which developers can use to report or treat Anomalies.

Severity Different anomalies have a different impact on a satellite. Some anomalies are just minor ones, which the onboard software can handle locally, e.g., by resetting a bus. Other ones are critical for the whole satellite, e.g., thermal Anomalies. Such critical anomalies might put the whole mission in danger. In order to categorize anomalies, Corfu provides this field that allows engineers to rate the severity of anomalies in advance. The software can react to the severity, eventually.

9.1.8 Telecommand

Telecommands are fixed commands, which the operation crew fires to achieve a stable operation and accomplish the mission goals. Engineers can define such Telecommands and their structure in the engineering model.

ID An integer number that identifies the telecommand in an Application. Indeed, this value must be unique among all the other telecommands in the application (the config parser checks this fact).

Name Each telecommand has a name; represented as a string. The name is not only used for debugging purposes and in the ground software; it is also the basis for the generated struct and method names.

Asynchronous This flag indicates whether the application shall handle the telecommand asynchronously. For all telecommands that have this flag set to true, the application invokes handling methods in the context of an extra command handling thread instead of the context of the communication middleware. Asynchronous handling is useful for telecommands that require some time for processing.

Most Telecommands carry some parameter values. The engineering model calls them Fields, which are referenced by Telecommands, taking their sequence into account.

9.1.9 Topic

A topic represents a communication channel between different applications. They have two elements in the engineering model

ID An integer number that identifies the telecommand in an Application. This value must be unique among all the other topics in the project (the config parser checks this fact).

Name Each topic has a name; represented as a string. The name is not only used for debugging purposes and in the ground software; it is also the basis for the generated struct, variable and method names.

Each topic is equipped with a data data type for the communication messages. Therefore, it references the struct entity.

9.1.10 Field

A Field represents a parameter or field in Telecommands, Telemetry, or other Structs. The green entities in figure 9.1 are those entities that build-up structure fields. A Field itself does not contain more than one parameter in the engineering model.

Name Each Field has a name; represented as a string. It has to be unique among the other fields of a Struct, Telecommand, or Telemetry (the config parser checks this fact). The name is not only used for debugging purposes and in the ground software; it is also the basis for the generated struct and method names.

Min Value This is the minimum value of the valid range. If this parameter is set, the onboard software automatically checks the value before writing it.

Min Value This is the maximum value of the valid range. If this parameter is set, the onboard software automatically checks the value before writing it.

A field can have different types. The references decide which types a field has. It can be either a plain PlainDataType, an Array, a BitArray, or a Struct.

9.1.11 PlainDataType

PlainDataType is the simplest data type for Fields and Arrays. It is a plain data structure that is available in C++, such as `uin32_t` or `float`. Its name just identifies the plain data type.

Name This is the C++ name of the plain data type.

9.1.12 Array

Arrays are a collection of the same data type with a fixed number of elements. Similar to Fields, the data type of array elements can be either a plain PlainDataType, an Array, a BitArray, or a Struct. The reference to a type decides which type the array elements have. In addition, Arrays have one parameter in the engineering model.

Length The number of elements the Array should hold.

9.1.13 Struct

In contrast to Arrays, Structs can hold fields with different data types. The Struct entity itself defines only one parameter in the engineering model.

Name This is the name of the struct, which the generator uses for the source code.

The struct references all the Fields that are part of its structure.

9.1.14 Telemetry

In the context of the engineering model definition, telemetry is very simple to telecommand. It also references fields that build up their payload structure.

ID An integer number that identifies the telemetry in an Application. Indeed, this value must be unique among all the other telemetry in the application (the config parser checks this fact).

Name Each Telemetry has a name; represented as a string. The name is not only used for debugging purposes and in the ground software; it is also the basis for the generated struct and method names.

9.1.15 SynchronousSubscription

In Corfu's engineering model, there are two types of subscriptions, DirectSubscription and AsynchronousSubscription. Both subscribe to topics, but they do it in different ways. DirectSubscriptions handle topic messages synchronously. That means they directly invoke a handling method executed in the context (thread) of the communication middleware. AsynchronousSubscriptions do not call a handling method; instead, it stores topic messages into a FIFO. The application has to read the FIFO to handle the messages eventually (in a thread). In the engineering model, DirectSubscriptions need no parameters; they only reference a topic they want to subscribe to.

9.1.16 AsynchronousSubscription

An AsynchronousSubscription connects a topic with a local FIFO in the application. For each AsynchronousSubscription, the generator creates an appropriate FIFO for the application. For this, the engineering model needs information about the length of the FIFO.

FIFO Length The number of elements that the FIFO shall be able to store.

Like for the DirectSubscription, the AsynchronousSubscription references the Topic that it shall subscribe to.

9.1.17 Publication

Defining Publications is very easy in Corfu's engineering model. It belongs to an Application and references a Topic for publishing. There is no parameter else in the engineering model.

9.1.18 SharedResource

Apart from the inter-application communication via topics, there are also cases in which applications share the same code and objects. The engineering model represents such objects as SharedResource. It just requires one parameter in the engineering model.

Name This value is the name of the class that the node shall instantiate.

The instantiated class is encapsulated into a `ThreadSafeData` class (see Section 17.1.1) to ensure thread-safety.

9.2 Behavioral Part of the Engineering Model

Apart from the software structure (described in the previous section), engineers can also define some behavioral aspects to describe some requirements of the user code. Figure 9.2 shows the configuration elements as entity relationship diagram. The elements on the left side define code elements, which the user has to implement. They all call a handler method whenever data arrives, or a thread iteration executes. Engineers can specify which response they expect from the user code. There are three response types: topic publications, anomaly reports, and telemetry emission. The static analyzer extracts information about these responses from the source code. Therefore, the framework can check whether the users actually implemented those responses.

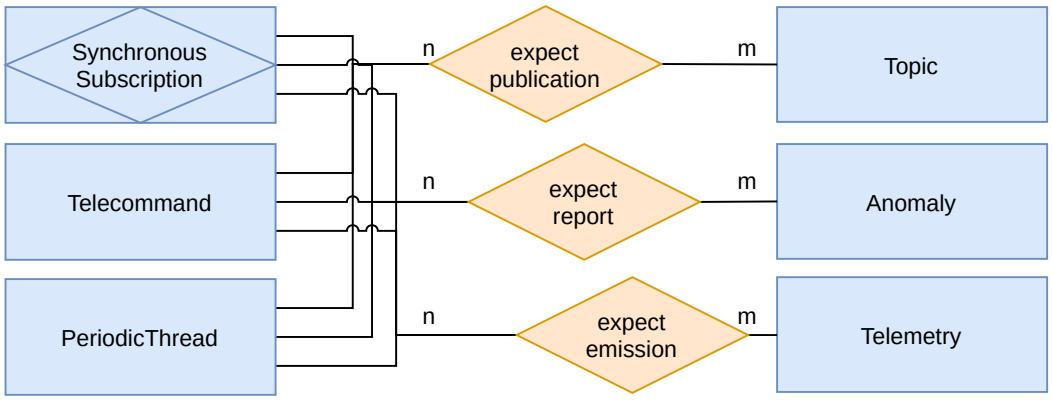


Figure 9.2.: The entity relationship diagram of behavioral aspects of the user code

10 Tools and Libraries of Corfu

Corfu comes with several libraries and tools. The following sections give introductions to them on a descriptive level; the part IV describes implementation details.

All the tools and libraries build on each other. Figure 10.1 shows the dependencies of the different tools and libraries to each other. For better clarity, the diagram does not show transitive usage; for example, corfu-ground-software uses parts of libcorfu-basic, which is not displayed there.

libcorfu-basic

The C++ standard library comes with a lot of valuable functions and classes[47, 91]. However, the standard implementation uses some features of C++, which are not desirable in safety-critical applications. For example, the JSF coding standard, as well as the coding directive of Rodos, forbid using exceptions. However, the classes and functions of the C++ standard library heavily rely on exceptions to notify applications of failures. Disabling exceptions and still using the standard library is not a good deal because of the missing error reporting. Our `libcorfu-basic` uses the error reporting approach of Rodos, which is a `Result` class, which either contains the return value or an error code (like in Rust[70]).

Another problem is the container classes of the C++ standard library. They use dynamic memory, which coding standards for safety-critical software prohibit. For that reason, we have created our own library like the standard library, which we call `libcorfu-basic`. The container classes use static memory. Hence, it directly allocates the required memory as a class member (e.g., directly on the stack) without a pointer to some allocated memory on the heap. It is not compatible with the standard library of C++ because of the different error reporting systems and template parameters for the container classes. Our library also does not implement all classes from the standard library.

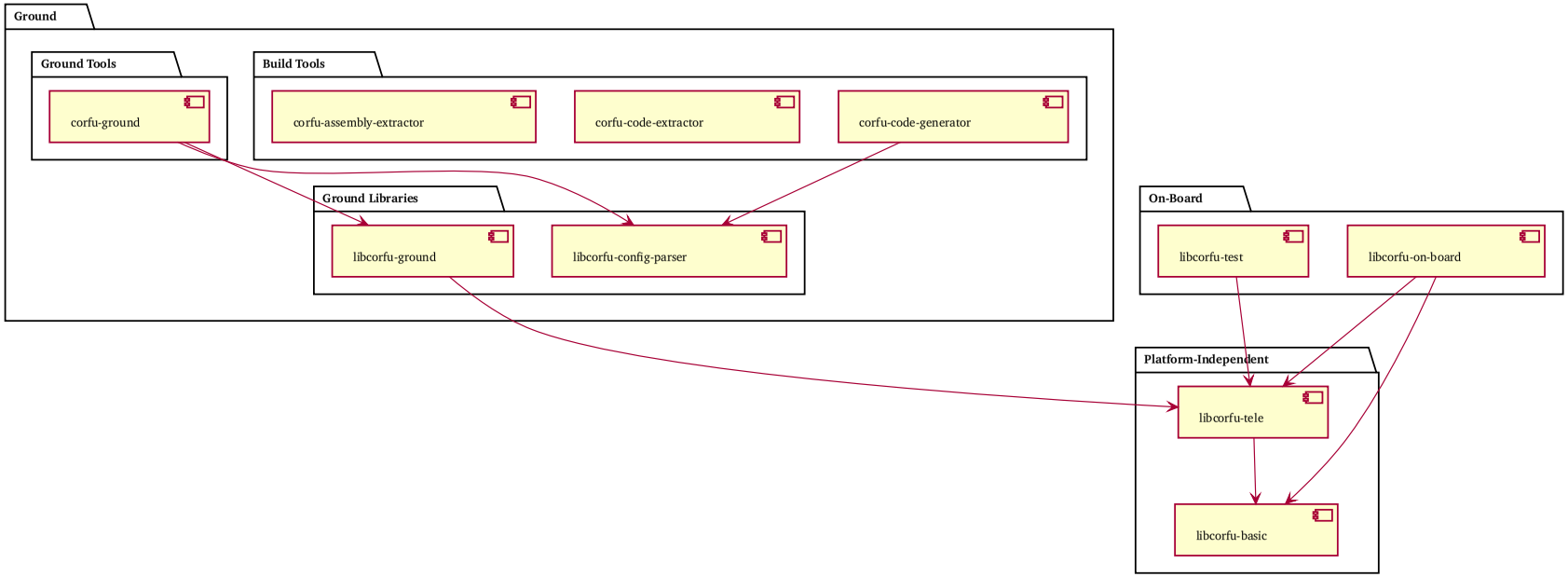


Figure 10.1.: Basic structure of Corfu's libraries and tools

libcorfu-tele

This library covers the communication between the ground and the space. Indeed, the physical layer of the communication, the radio technology, depends on the mission. For example, some use S-Band, and others use HAM radio. The selection of radio technology sometimes also influences the second OSI layer, the data link layer. For example, HAM radio often requires an open protocol here, the AX.25 protocol.

However, the encoding above the second layer is always the same — this is what `libcorfu-tele` defines. This library comes with the structure definitions of telecommands and telemetry and serializing functions, which both share, onboard, and ground software.

libcorfu-on-board

This library contains generic code for the onboard software. It is part of all onboard software created with Corfu. That means it is independent of the software's configuration. It contains generic base classes for applications and nodes, declarations of common topics and data structures, and helper functions, e.g., for thread synchronization.

libcorfu-ground

Corfu provides does not only support developing onboard software. Furthermore, it is a complete software suite, which also covers the ground part. The ground library is the counterpart to the onboard software. For the communication between the ground and the space, this library encodes and decodes telecommand and telemetry packets. This library is useful for developing custom ground software that perfectly integrates into our framework. Corfu comes with a reference implementation of ground software, which relies on this library (see below).

corfu-code-generator

This tool generates C++ source code based on the model of the software. It combines information from the model with template files for the source code. The generator creates an abstract class for each application, which handles local telecommand distribution, topic subscription, and thread handling. For each action, it creates a

pure virtual function for the user to implement in a subclass. Those actions include the execution code of threads, handling functions for telecommands, and direct topic subscriptions. See section 11.5 for implementation details of the code generator.

libcorfu-config-parser

This library is responsible for parsing the model from configuration files. It checks the configuration code syntactically and semantically. For the user, it provides the model in c++ data structures with easy access. Every tool that has to process the model uses this library.

corfu-ground

This tool is the reference implementation of ground software using Corfu. It relies on Corfu's ground library to access the onboard software model and the communication interface. Based on the model, it dynamically creates a graphical user interface based on Qt[1]. The graphical interface displays the current values of standard telemetry and provides an easy way to send telecommands and display information from extended telemetry.

corfu-assembly-extractor

We have two different tools for extracting information from the final software: the corfu-assembly-extractor and the Corfu-code-extractor. The corfu-assembly-extractor works on the assembly level in order to extract information such as stack usage. See section 12.2 for implementation details of the assembly extractor.

corfu-code-extractor

The Corfu-code-extractor extracts information on the source code level. It parses the source code, builds the abstract syntax tree (AST), and extracts information from it. The library saves the resulting extended model along with the engineering model. See section 12.3 for implementation details of the code extractor.

[1] https://www.qt.io/

libcorfu-test

This library provides a test environment for unit and integration tests. Developers have to write unit tests to cover their manually written code. They only test the manual code, no other code of the software stack. In order to test the complete code, developers have to create integration tests. Those tests use the ground-space communication interface, namely telecommands and telemetry, for testing. Developers define telecommands to send and which telemetry data to expect. Integration tests can also be executed directly on the target platform. Chapter 14 goes into more detail about testing the onboard software.

Part IV

Implementation of Corfu

11 Configuration Files and Generated Code

Chapter 9 has introduced the engineering model in a theoretical way. This chapter shows how we practically encode the engineering model into configuration files.

In the field of modeling, there are different ways to encode engineering models — see also chapter 3. We have decided to use YAML as the file format for defining the engineering model of onboard software. It is a format with straightforward syntax. YAML lets users define a nested data structure of three element types: associative lists, arrays, and scalar types. In contrast to XML, YAML has nearly no boilerplate code, making it easier to write manually.

11.1 Directory Structure for Satellite Projects

Similar to the hierarchical division of space missions, Corfu configures onboard software hierarchically. Figure 11.1 shows an exemplary folder structure of a Corfu project. All the configuration files have the `yml` suffix in their filename. They contain the description of the engineering model in the YAML file format. At the top-level directory, there is the project.yml file, which contains configuration about the whole project. There are three folders: apps, nodes, and generated. The apps folder contains a subfolder for each application containing the configuration and user code of the applications in the project. Each application has its configuration file describing its structure and communication interface. Along with the configuration file, there are two folders for each application: the source folder containing the application code and the test folder containing corresponding unit tests. Those applications can be part of nodes, which engineers define in the nodes folder. In contrast to applications, nodes do not require user code. Therefore, there is only a configuration file for each node and no source code.

Finally, there is a generated folder, which contains the code that Corfu generates. This folder is read-only for users. This folder's content will be re-generated several

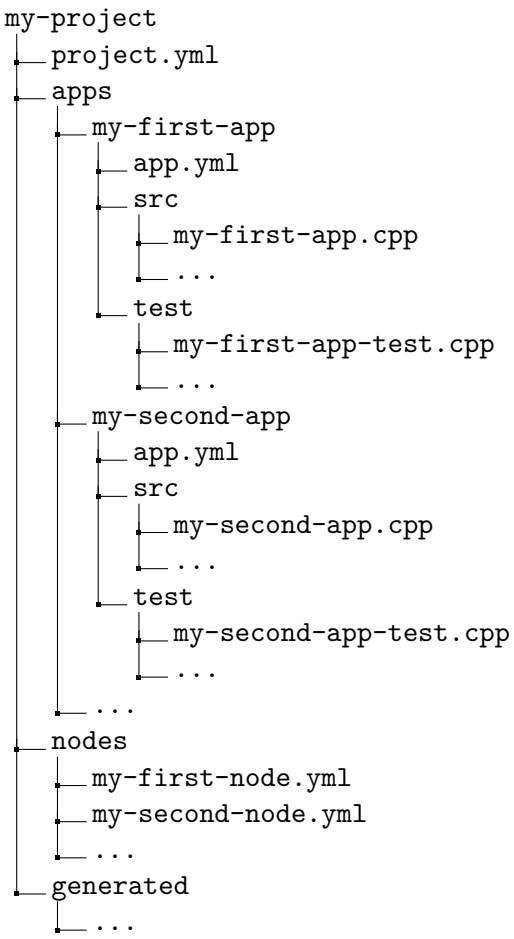


Figure 11.1.: Directory structure of a OBSW project using Corfu

times because of the iterative approach. The following sections present the different elements of configuration files.

11.2 Project Configuration File

The global configuration file at the root folder of the user source code defines common topics. Several applications can use those topics in order to exchange data. Consequently, such topics do not "belong" to a particular application and, therefore, they have to be defined globally. Topics are the only items defined in the global configuration file. The listing 11.1 shows an example configuration file for a project.

```yaml
---
name: myProject
topics:
  topicA: uint8_t
  topicB: uint16_t
  topicC:
    fields:
      fieldA: uint8_t
      fieldB: float
```

Listing 11.1: Example project configuration in YAML

The project does not have its own classes or structures in the source code. It defines only the project's name that either become available in the onboard code.

11.2.1 Name

The `name` parameter is just a string containing the project's name to have it available, e.g., for generating documentation files. Its value is globally available in the project, e.g., for debugging purposes. However, in orbit, this value is rarely of use. Hence, we rely on compiler/linker optimizations to remove the value if it is unused. The listing 11.2 shows that the generated source code is just a constant string in the Corfu namespace.

```
namespace corfu::project {
  const char *const NAME = "myProject";
}
```

Listing 11.2: Example generated code for the project's name

11.2.2 Topics

Even if engineers define topics at the project level, the generator creates code on the node level. There, it generates only those topics that the node's applications use. If the topic's data type is a custom struct, the generator will create a corresponding C++ `struct`. In addition, the node receives an instance of the topic for usage in the applications. The listing 11.3 shows exemplary generated topic code.

```
struct TopicCPayload {
  uint8_t fieldA;
  float fieldB;
```

```
};

// in the node classes
Topic<TopicCPayload> topicC;
```

Listing 11.3: Example generated code for a topic with custom data type structure

11.3 Application Configuration File

Each application has its own configuration file that describes different aspects of the application. The listing 11.4 shows an example configuration of applications.

```
---
name: MyApp
id: 137
compileTimeParameters:
  ParamA:
    type: int32_t
    defaultValue: 1337
runTimeParameters:
  paramB:
    type: int32_t
    defaultValue: 1338
    telecommandId: 64
    valueRange:
      min: 100
      max: 2000
  mySecondRunTimeParameter:
    type: int16_t
    defaultValue: 1339
telecommands:
  telecommandA:
    id: 2
  telecommandB:
    id: 3
    fields:
      myField: uint8_t
extendedTelemetry:
  myTelemetry:
    id: 1
    fields:
      fieldA: uint8_t
```

```
      fieldB: float
standardTelemetry:
  myIntField: uint16_t
  myFloatField: float
threads:
  myPeriodicThread:
    firstRun: 2_s
    period: 1_s
    priority: 100
    maxStackSize: 1_kB
subscribe:
  topicA:
    fifo: 8
  topicB: synchonous
publish:
  - myThirdTopic
```

Listing 11.4: Example app configuration in YAML

As already described in the conception section 8.3, the code generator creates specific classes based on the applications' configurations. Those classes inherit from the generic `corfu::App` as the listing 11.5 shows.

The generated code represents the compile-time parameters as template parameters. This allows the nodes to pass custom values, which are usable at compile-time in the application user code. The generator creates all the application components within the class body, like the rest of this section presents.

```
namespace generated {
  class MyApp<ParamA> : public corfu::App {
    // ...
  }
}
```

Listing 11.5: Example generated code for an app

11.3.1 Name

The `name` parameter is just a string containing the application's name to have it available, e.g., for generating documentation files. Its value is globally available in the application, e.g., for debugging purposes. However, in orbit, this value is rarely of use. Hence, we rely on compiler/linker optimizations to remove the value if it is

unused. The listing 11.6 shows that the generated source code is a constant string in the generated application class.

```
1 const char *const NAME = "MyApp";
```

Listing 11.6: Example generated code for the application's name

11.3.2 ID

Each application contains an ID used on board to address the application for telecommand destinations and telemetry sources. The generic `corfu::App` class performs the generic telecommand handling. Therefore, the ID value is passed to the generic class in the constructor, as the listing 11.7 shows.

```
1 MyApp() : corfu::App(137) {}
```

Listing 11.7: Example generated code for the application's ID

11.3.3 Compile Time Parameters

Compile-time parameters describe constant values in the source code, which the code cannot at run-time. Consequently, the generated code consists of `constexprs` as the listing 11.8 shows. The value in the generated code depends on the instantiation in the nodes. In the node configuration, such compile time parameter values can be overridden (see section 11.4). If there is no overriding value, the generator uses the default value from the application configuration.

To enable the overriding, we have to split the declaration from the definition. Therefore, the header file of the generated application code contains the declaration. There are separate source code files for the nodes, which define the value of the parameters eventually.

```
1 // header
2 extern constexpr int32_t paramA;
3
4 // node-dependent source
5 constexpr int32_t generated::my_app::paramA = 1337;
```

Listing 11.8: Example generated code for the application's compile-time parameter

11.3.4 Run Time Parameters

In contrast to compile-time parameters, run-time parameters can be changed at run-time, i.e., in orbit. The telecommand ID is optional. If it has a value, the code generator automatically creates an appropriate telecommand for changing the value. It does nothing more than setting the value and checking the range if set.

Run-Time parameters can have a range with a minimum and maximum value. If this is the case, the generated code encapsulates the variable in a `RangedValue` class in the generated code. The listing 11.9 shows exemplary generated code.

```
1  int32_t ProtectedVariable<RangedValue<int16_t>> paramB{1338};
2
3  ErrorCode handleParamBUpdate(const int32_t &newValue) {
4    return paramB->lock()->updateValue(newValue);
5  }
```

Listing 11.9: Example generated code for the application's run-time parameter

11.3.5 Telecommands

Telecommands represent interaction points for the ground software. On the satellite side, the `handleTelecommand` method deserializes the telecommand payload and invokes a particular method. The specific handling method is declared pure virtual; it has to be implemented by the user in the subclass. The listing 11.10 shows exemplary generated code for handling telecommands in applications.

```
1  virtual Error handleTelecommandA() = 0;
2  virtual Error handleTelecommandB(TelecommandBPayload &payload
       ) = 0;
3
4  Error handleTelecommand(Telecommand &telecommand) {
5    if(telecommand.appId != appId) { return NOT_FOR_ME; }
6
7    switch(telecommand.telecommandId) {
8      case 2:
9        return handleTelecommandA();
10
11     case 3:
12       TelecommandBPayload payload;
13       TelecommandBPayload.deserialize(telecommand.
             getSerializedPayload());
14       return handleTelecommandB(payload);
```

```
15    }
16  }
```

Listing 11.10: Example generated code for the application's telecommand handling

11.3.6 Standard Telemetry

Applications contribute some fields to the standard telemetry of the nodes. For each application, the code generator creates its own struct. In the generated application class, there is an instance of the struct used when collecting the standard telemetry from each application. In best practice, users keep this instance up to date to have it always available for standard telemetry. For a detailed description of how standard telemetry works, have a look at section 8.4.2. The listing 11.11 shows exemplary generated code for standard telemetry in apps.

```
1  struct StandardTelemetry : Serializable {
2    uint16_t fieldA;
3    float fieldB;
4
5    Result<size_t> serialize(Slice<uint8_t>& slice) override;
6    Result<size_t> deserialize(Slice<uint8_t>& slice) override;
7  };
8
9  corfu::ProtectedVariable<StandardTelemetry> stdTM;
```

Listing 11.11: Example generated code for the application's standard telemetry

11.3.7 Extended Telemetry

Applications can define other telemetry types called extended telemetry. In contrast to standard telemetry, extended telemetry is usually not automatically sent but often used to respond to telecommands. They provide a way to report additional information to the ground.

Also, here, the code generator creates a struct with the telemetry payload, as the listing 11.12 shows.

```
1  struct MyExtendedTelelemetry : TelemetryPayload {
2    uint8_t fieldA;
3    float fieldB;
4
```

```
  Result<size_t> serialize(Slice<uint8_t> &slice) override;
  Result<size_t> serialize(Slice<uint8_t> &slice) override;
};
```

Listing 11.12: Example generated code for the application's extended telemetry

11.3.8 Threads

Applications can define threads, which represent active execution paths that the operating system schedules. There are two different types of thread: periodic and aperiodic threads. In the configuration, periodic threads require timing parameters. If those values are not defined, the generator creates an aperiodic thread.

```
virtual void runMyPeriodicThread() = 0;

class MyPeriodicThread : public PeriodicThread<1_kB> {
  MyApp& app;

 public:
  explicit MyPeriodicThread(ExampleApp& app)
    : PeriodicThread<1_kB>(1_s, 1_s, "myThread", 100),
      app(app) {}

 protected:
  void runIteration() override { app.runMyThread(); }
} myPeriodicThread{ *this };
```

Listing 11.13: Example generated code for the application's periodic threads

For each periodic thread, the code generator creates an own subclass of `corfu::PeriodicThread`, which invokes a specific pure virtual function periodically according to the timing parameters. The listing 11.13 shows exemplary generated code for periodic threads.

11.3.9 Topic Subscription

The software uses topics for inter-application communication. Hence, applications can subscribe and publish to topics. In the subscription part of the configuration file, applications define which topics they want to subscribe to. There are two ways of subscribing to topics: synchronously and asynchronously. The first directly calls a handling function upon receipt of a topic message as the listing 11.14 shows.

```
virtual void handleTopicA(const uint8_t &message) = 0;

class TopicAReceiver : public SubscriberReceiver<uint8_t> {
  MyApp& app;

 public:
  explicit TopicAReceiver(MyApp& app)
    : SubscriberReceiver<uint8_t>(topicA, "MyApp"),
      app(app) {}

  void put(uint8_t& message) override {
    app.handleTopicA(message);
  }
} topicAReceiverSubscriber{ *this };
```

Listing 11.14: Example generated code for the application's synchronous topic subscriptions

The handling function is defined to be purely virtual. Hence, developers have to implement the message consumption in the subclass. The asynchronous way directly saves topic messages into a FIFO as the listing 11.15 shows.

```
SyncFifo<uint16_t> topicBFifo;
Subscriber topicBSubscriber(topicB, topicBFifo);
```

Listing 11.15: Example generated code for the application's asynchronous topic subscriptions

Developers have to make sure that all the values saved in the FIFO have to be processed (by a thread) eventually.

11.3.10 Topic Publication

In the configuration file, topic publications are just a list of topics the application intends to publish. Each topic in an application's publication list is made available to the application's source code, as the listing 11.16 shows.

```
extern Topic<float> topicC;
```

Listing 11.16: Example generated code for the application's topic publication

11.4 Node Configuration File

In the `nodes` directory, each node has its own configuration file. The listing 11.17 shows an example configuration of a node.

```
1 --
2 name: MyNode
3 id: 200
4 apps:
5   myApp:
6     myCompileTimeParameter: 2669
7   myOtherApp: default
```

Listing 11.17: Example node configuration in YAML

Like applications, the code generator creates a specific class according to the configuration, which inherits from a generic class from Corfu — like the listing 11.18 shows.

```
1 class MyNode : public corfu::Node {
2   // ...
3 }
```

Listing 11.18: Example generated code for a node

Nodes do not need any manual code; all information required for nodes is already available in the configuration of nodes and applications.

11.4.1 Name

The `name` parameter is just a string containing the node's name to have it available, e.g., for generating documentation files. Its value is globally available in the node, e.g., for debugging purposes. However, in orbit, this value is rarely of use. Hence, we rely on compiler/linker optimizations to remove the value if it is unused. The listing 11.19 shows that the generated source code is a constant string in the generated node class.

```
1 const char *const NAME = "MyNode";
```

Listing 11.19: Example generated code for the node's name

11.4.2 ID

Each node contains an ID, which the onboard software uses to address the application for telecommand destinations and telemetry sources. Therefore, the ID value is passed to the generic class in the constructor, as the listing 11.20 shows.

```
1 MyNode() : corfu::Node(200) {}
```

Listing 11.20: Example generated code for the node's ID

11.4.3 Apps

In the configuration, `apps` contains a list of applications that the node shall instantiate. Here, nodes can override the values of compile-time parameters by providing a new value. If no parameter values shall be overridden for an application, developers must define `default` to keep the default values. The node class instantiates all the defined applications as member variables as the listing 11.21 shows. If applications have compile-time parameters, the node passes their (overwritten) value as template parameter.

```
1 ::MyApp myApp<2669>;
2 ::MyOtherApp myOtherApp;
```

Listing 11.21: Example generated code for the node's list of applications

11.5 Code Generation Process

Based on the model, a generator creates various artifacts, e.g., software images or documentation documents. This chapter presents implementation details of the automatic generation process.

The source code generator uses two libraries to produce source code and documentation:

libcorfu-config-parser This library parses and validates the configuration files, i.e., the model. It provides the model as a C++ representation to the generator.

inja[1] This is an external library. It is a templating engine similar to Python's Jinja[2] [2].

[2] https://palletsprojects.com/p/jinja/

Figure 11.2 shows the process of generating source code. The config parser library delivers a ready C++ representation of the entire model. If the config files are valid, the generator converts their information is parsed into representation that serves as input to inja.

11.5.1 The Template Files

Inja's template files are extensions of the desired output. Listing 11.22 shows an example template file for creating struct in C++ based on information from the model / configuration files. Developers can write placeholders into template files, which inja handles. Those placeholders implement different features. The simplest one is just replacing the placeholder with a variable's value — see the first line in the example. There, the name is passed to a function named `CamelCase`, which Corfu's generator defines. Its purpose is to reformat the passed string into an upper camel case format, e.g., `my-struct` to `MyStruct`. The resulting value between `{{` and `}}` is directly printed into the generated file.

```
1 struct {{ CamelCase(name) }} {
2   {% for field in fields %}
3     {% if field.type.isArray %}
4       {{ field.type.array.type.name }} {{ field.name }}[{{
            field.type.array.length }}];
5     {% else %}
6       {{ field.type.name }} {{ field.name }}{};
7     {% endif %}
8   {% endfor %}
9 };
```

Listing 11.22: Example template for generating a struct

More complex placeholders provide control structures such as if/else and loops. They are enclosed into `{%` and `%}`. Like in line 2, loops repeat their body and inja prints the outputs of all iterations consecutively. In the example, inja iterates through all the fields to print them into the generated file. For each field, inja checks whether it is an array or not. If it is an array, inja prints the member variable as an array, i.e., with square brackets defining the array's length. Otherwise, inja creates a standard non-array member variable. Listing 11.23 shows an example struct that had been generated with the template from listing 11.22.

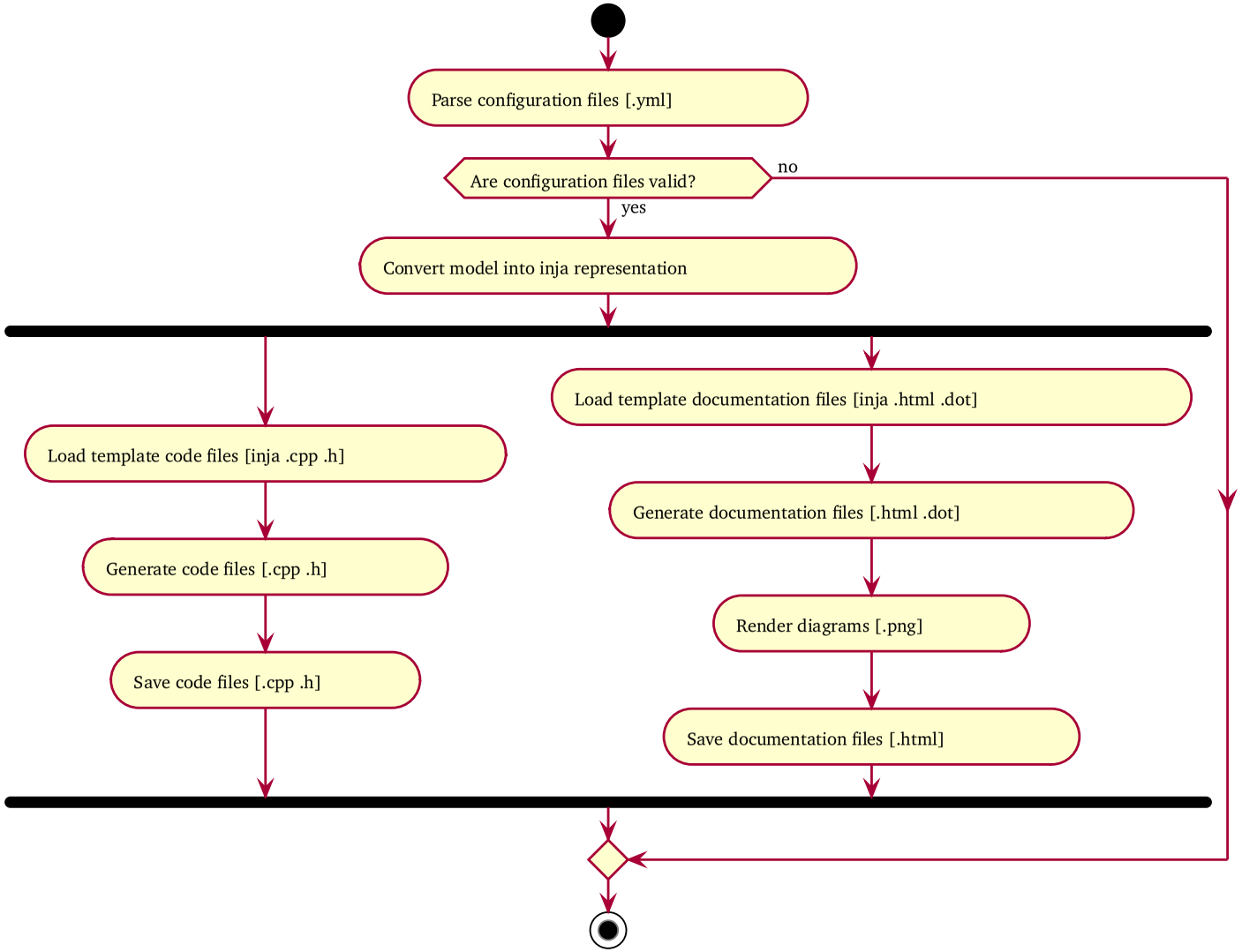


Figure 11.2.: The activity diagram of the source code generation process

```
struct MyStruct {
  int32_t myFirstField;
  int16_t mySecondField[8];
};
```

Listing 11.23: Example template for generating a struct

11.5.2 Documentation Generation

Apart from the obligatory generation of code, Corfu also generates documentation files. Those documentation files contain information from the (extended) model and display it vividly and interactively in the browser. The generator creates pages for each application, node, and topic, which contain all the information from the model. In addition, the generator creates diagrams displaying relationships of different components. For example, there is a diagram about the publishing and subscribing applications on topic pages. Figure 11.3 and 11.4 show screenshots of generated documentation. The first one contains a diagram that shows which topics the example application subscribes. The second one shows the structure of an example telecommand.

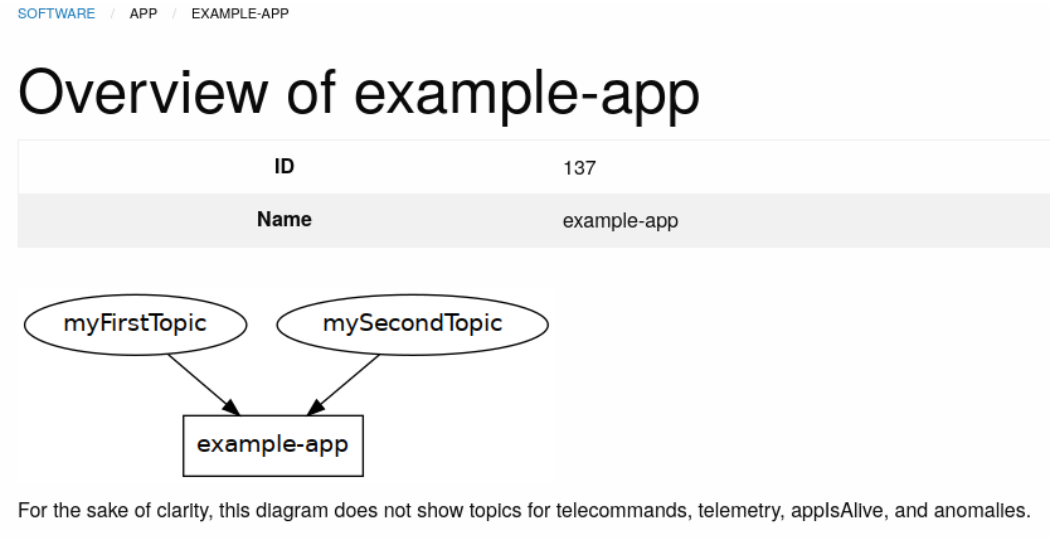


Figure 11.3.: Example of generated documentation with a topic diagram

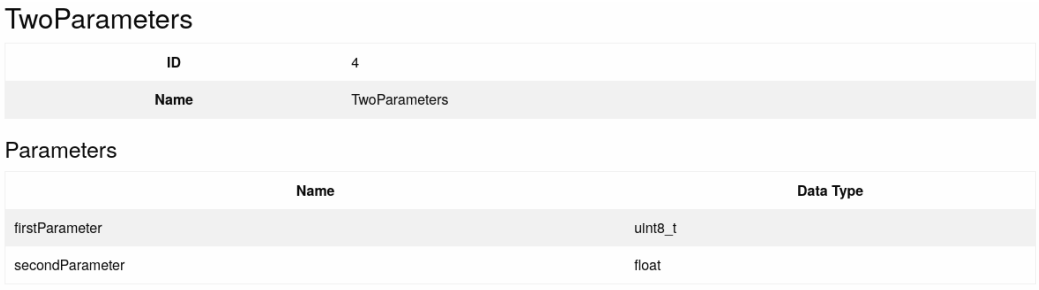


Figure 11.4.: Example of generated documentation with a telecommand

12 Feedback from User Code to the Model

The classical model-driven approach knows only one direction of information: from the requirements into the model and from the model into the source code. However, models are abstract; therefore, code generators can only create a part of the source code from the model. Hence, models do not contain every detail of the final source code. Developers manually write the other part. Implementation details influence the total system, e.g., memory and CPU usage. Therefore, it is helpful to extract information from the user-written source code and extend the engineering model. Extracting information and extending the model is our feedback approach, which this chapter presents.

12.1 The Extended Model

Chapter 9 described the structure of the engineering model. Developers and engineers manually define the engineering model in configuration files (see chapter 11).

Our feedback approach extracts information from the code in two ways: binary analysis and user-written source code analysis. The analyzing tools collect the extracted information in a new model, which we name extracted model. The extracted model introduces new entities towards the engineering model. Hence, elements of the extracted model directly reference elements of the engineering model. In total, we call the combination of both models (predefined and extracted model) the extended model — see figure 12.1.

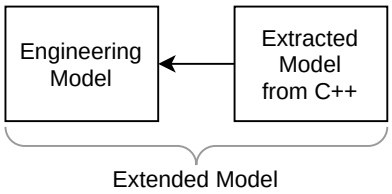


Figure 12.1.: The relations of engineering, extracted, and extended models

The extracted model contains different types of information, which it links with entities of the engineering model. Figure 12.2 shows an entity relationship diagram of the extracted model. All the yellow entities are entities from the engineering model. The new entities from the extracted model reference those entities from the engineering model. Apart from the yellow entities, we have nodes with two different colors, blue and green. The colors represent two independent sets of data, which we use to implement different features in the software.

The green entities contain information about the source code. It contains all the functions (or methods) defined in the source code and their relation to calls and overrides. In addition, the extracted model comprises information about which functions publish data to topics or emit extended telemetry. For example, section 13.3 describes how this information is used for stack usage analysis.

The blue entities contain information about events in the code. Those events consist of a message string, a severity, and optional parameters. Section 15 describes how this information is used.

Corfu extracts all the information (blue and green) from the compiled binary file and the source code. It saves the extracted model into a SQLite database. The following two sections describe how Corfu extracts the information.

12.2 Assembly Analysis

Some information can be easier extracted from the assembly than from the source code. One example is the stack usage of functions. The stack utilization highly depends on compiler optimizations. For example, unused local variables might be removed, which, hence, do not land on the stack. Hence, we extract stack usage information from the assembly. For the analysis of stack usage of functions, Corfu uses the same approach as the `checkstack.pl` script from the linux kernel[1].

The basis is an assembly of the compiled file. Listing 12.2 shows an example assembly that has been generated from the function of listing 12.1. We have used clang 11.1.0 with disabled optimizations (`-O0`). The example function (listing 12.1) does nothing else than just using some stack memory. The assembly code shows that the stack pointer is decremented once in line 4 and incremented once in line 13. Hence, we parse those lines in order to determine the maximum stack usage of the function.

[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/scripts/checkstack.pl

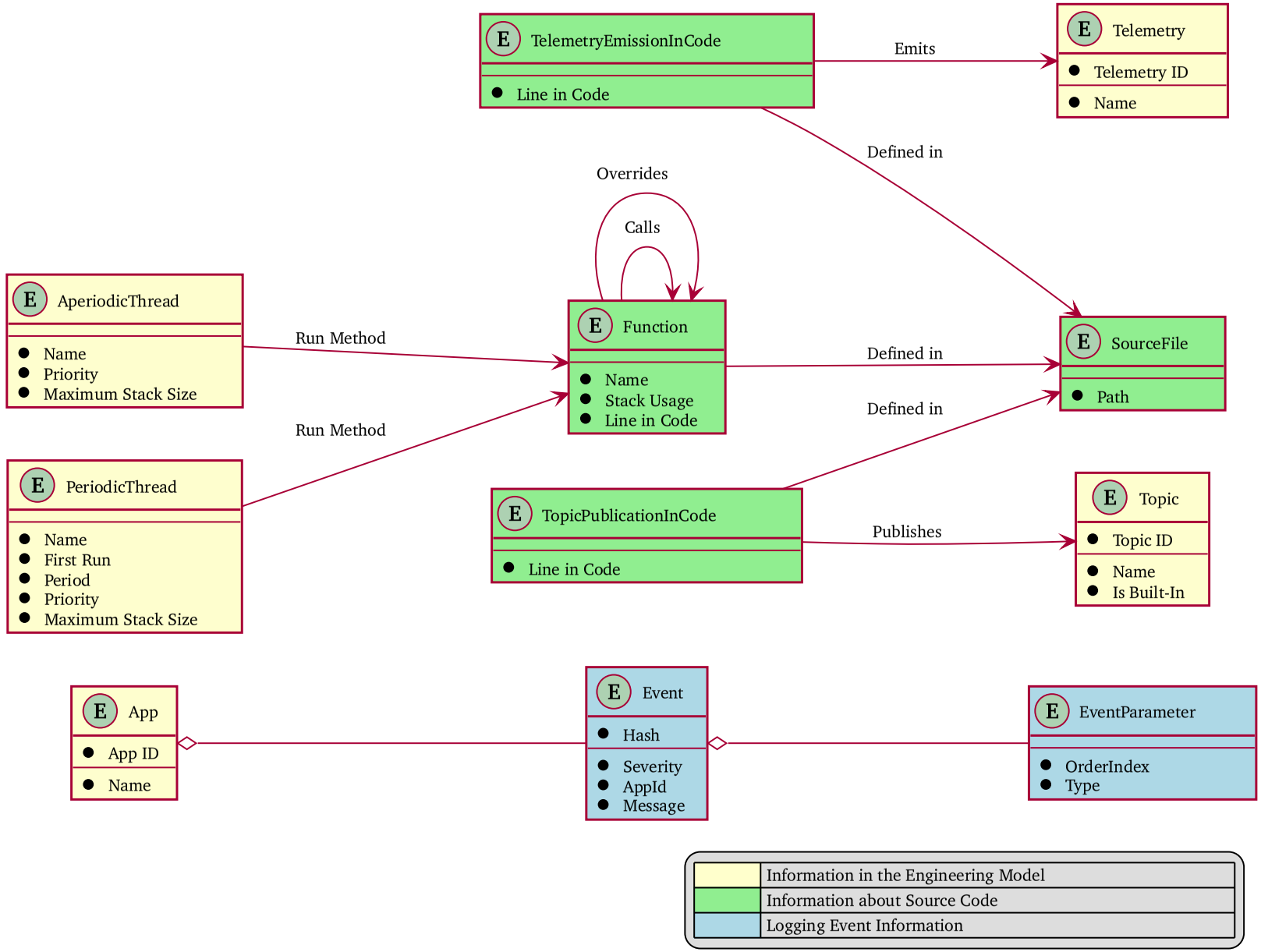


Figure 12.2.: The extended model

```
void myFunction() {
  char arr[8];
}
```

Listing 12.1: Example function for binary analysis

```
0000000000401190 <_Z10myFunctionv>:
  401190: 55                    push   %rbp
  401191: 48 89 e5              mov    %rsp,%rbp
  401194: 48 83 ec 10           sub    $0x10,%rsp
  401198: 64 48 8b 04 25 28 00  mov    %fs:0x28,%rax
  40119f: 00 00
  4011a1: 48 89 45 f8           mov    %rax,-0x8(%rbp)
  4011a5: 64 48 8b 04 25 28 00  mov    %fs:0x28,%rax
  4011ac: 00 00
  4011ae: 48 8b 4d f8           mov    -0x8(%rbp),%rcx
  4011b2: 48 39 c8              cmp    %rcx,%rax
  4011b5: 0f 85 06 00 00 00     jne    4011c1 <
     _Z10myFunctionv+0x31>
  4011bb: 48 83 c4 10           add    $0x10,%rsp
  4011bf: 5d                    pop    %rbp
  4011c0: c3                    retq
  4011c1: e8 6a fe ff ff        callq  401030 <
     __stack_chk_fail@plt>
  4011c6: 66 2e 0f 1f 84 00 00  nopw   %cs:0x0(%rax,%rax,1)
```

Listing 12.2: x86 Assembly of the example function for binary analysis

By applying regular expression, the assembly analyzer detects three different lines in the assembly file: function labels, increasing, and decreasing the stack pointer's value. Our tool detects those lines by applying regular expressions (like the tool from the linux kernel). Listing 12.3 shows those three regular expressions. These regular expressions are directly used in an algorithm that adds up the maximum stack usage for each function — see algorithm 1.

```
static const std::regex functionRegex{"^[0-9a-f]+ <(.+)>:$"};

static const std::regex stackIncreaseRegex
  {"^.*sub    \\$(0x[0-9a-f]{1,8}),\\%(e|r)sp$"};

static const std::regex stackDecreaseRegex
  {"^.*add    \\$(0x[0-9a-f]{1,8}),\\%(e|r)sp$"};
```

Listing 12.3: Regular expressions for analyzing x86 assembly code

Algorithm 1 Determining maximum stack usage of functions

```
functionMap ←{}
for line ∈ file do
    if line matches functionRegex then
        if functionName is not empty then
            functionMap[functionName] ←maximumStackSize
        end if
        functionName ← functionRegex(1)
        stackSize ←0
        maximumStackSize ←0
    else if line matches stackIncreaseRegex then
        stackSize ←stackSize + stackIncreaseRegex(1)
        if stackSize > maximumStackSize then
            maximumStackSize ← stackSize
        end if
    else if line matches stackDecreaseRegex then
        stackSize ←stackSize - stackDecreaseRegex(1)
    end if
end for
if functionName is not empty then
    functionMap[functionName] ←maximumStackSize
end if
```

The biggest drawback of analyzing assembly instead of source code is that it is not independent of the hardware. For each compilation platform, the analyzer has to come with its own assembly analyzer module. However, relying on the source code comes with more inaccurate results because the compiler applies code optimization, such as removing unnecessary variables or function inlining. In addition, the selected ABI and variable sizes differ along with the compiler and the target hardware, which analyzers cannot read from the source code.

12.3 Source Code Analysis

Some information can be better extracted from the source code. In some cases, it is also necessary to read the information from the source code. With enabled optimization, the compiler may inline functions that do not exist as separate functions in the assembly and binary.

However, parsing C++ is very complex. The C++ standard is 1.815 pages long in the current standard[47]. Writing a completely new parser from scratch is a monumental task. Therefore, we rely on an existing parser: Clang.

Originally, Clang was an input frontend for the compiler LLVM. It enables LLVM to process the C programming language group, including C and C++[57]. Clang first generates the abstract syntax tree (AST) from the source code and further processes it for optimization and compilation. Fortunately, Clang provides a rich programming interface to access intermediate representations of the source code, including the AST. This is used by several tools[94, 99, 101]. There are different ways to access the internals of Clang. For example, one could create plugins that LLVM executes when compiling source code. Another access option is LibTooling[58], a standalone library, which allows arbitrary projects to parse C++ code and access the AST. Corfu's code analyzer uses the latter, LibTooling, investigating the user source code.

Clang provides different types of introspecting the AST. One can manually or automatically traverse the AST from top to bottom. Another approach, which Clang provides, is ASTMatchers. Here, users define patterns within the AST and pass them to clang. Whenever Clang finds the pattern, it invokes a callback function with the found subtree as a parameter.

12.3.1 Clang's AST Representation

Apart from the source code's syntactical structure, Clang also includes references that describe semantic links between nodes of the AST. Figure 12.3 shows an AST for the file `test.cpp` representing the following source code of listing 12.4.

```
1  class Test {
2  public:
3    bool var;
4    void methodA();
5    void methodB();
6    void methodC();
7  };
8
9  void Test::methodA() {
10   methodC();
11   if(var) { methodB(); }
12 }
13 void Test::methodB() {
14   methodC();
15 }
16 void Test::methodC() { }
```

Listing 12.4: Example source code for abstract syntax tree demonstration

The type of the root node is always `TranslationUnitDecl`. Due to `#include` statements in the source file, Clang adds nodes from additional files, usually header files, to the AST. In the example, the AST contains nodes from a source and a header file; figure 12.3 separates the nodes by their origin (dashed boxes). The header file declares the class `Test` (node `CXXRecordDecl`[2]), the member variable `var` (node `FieldDecl`) and the three methods `methodA`, `methodB`, and `methodC` (nodes `CXXMethodDecl`). The implementations (definitions) of the methods are written in the source file and not the header file. Therefore, the method declaration nodes `CXXMethodDecl` from the header file lack syntactic children (black edges). Instead, they have semantic children (blue edges) that describe their implementations (definitions). The implementations (definitions) of the methods are syntactical children of the root node (`TranslationUnitDecl`).

Curly braces always enclose a method's body; the `CompoundStmt` stands for this enclosed compound of statements. Consequently, the children of `CompoundStmt` describe the method's body. Three subsequent nodes define a method call. There, `CXXMemberCallExpr` specifies the return type, `MemberExpr` describes the member name (method or variable), and `CXXThisExpr` defines the variable (object) on which the call is executed. Additionally, Clang references the declaration of the called method to the `MemberExpr` nodes (green edges). Moreover, if a method is already declared before, Clang references the `CxxMethodDecl` to the previous declaration node (also `CxxMethodDecl`, red edges). The `methodA` calls one or two other methods; `methodC` is always called, whereas the call of `methodB` depends on the evaluation result of the if statement. The left child of the node `IfStmt` defines the condition part, and the right child defines the *then part*. Thus, the call of `methodB` is only executed if the value of `var` is `true`.

The resulting structure is not a tree regarding the extra (colored) edges; instead, it is a directed acyclic graph (DAC).

12.3.2 Extracting with ASTMatchers (Example: Events)

The analyzer extracts the information for the extended model through the AST (abstract syntax tree) generated by clang. The libtooling interface of clang comes with `ASTMatchers`, which allow defining filters over the AST to find specific patterns. We depict the pattern of finding an event within the source code in figure 12.4. The string on the right side contains the string message used for the event. To retrieve the types of the event parameters, the analyzer iterates over the remaining

[2]The prefix CXX in the types of the nodes stands for C++.

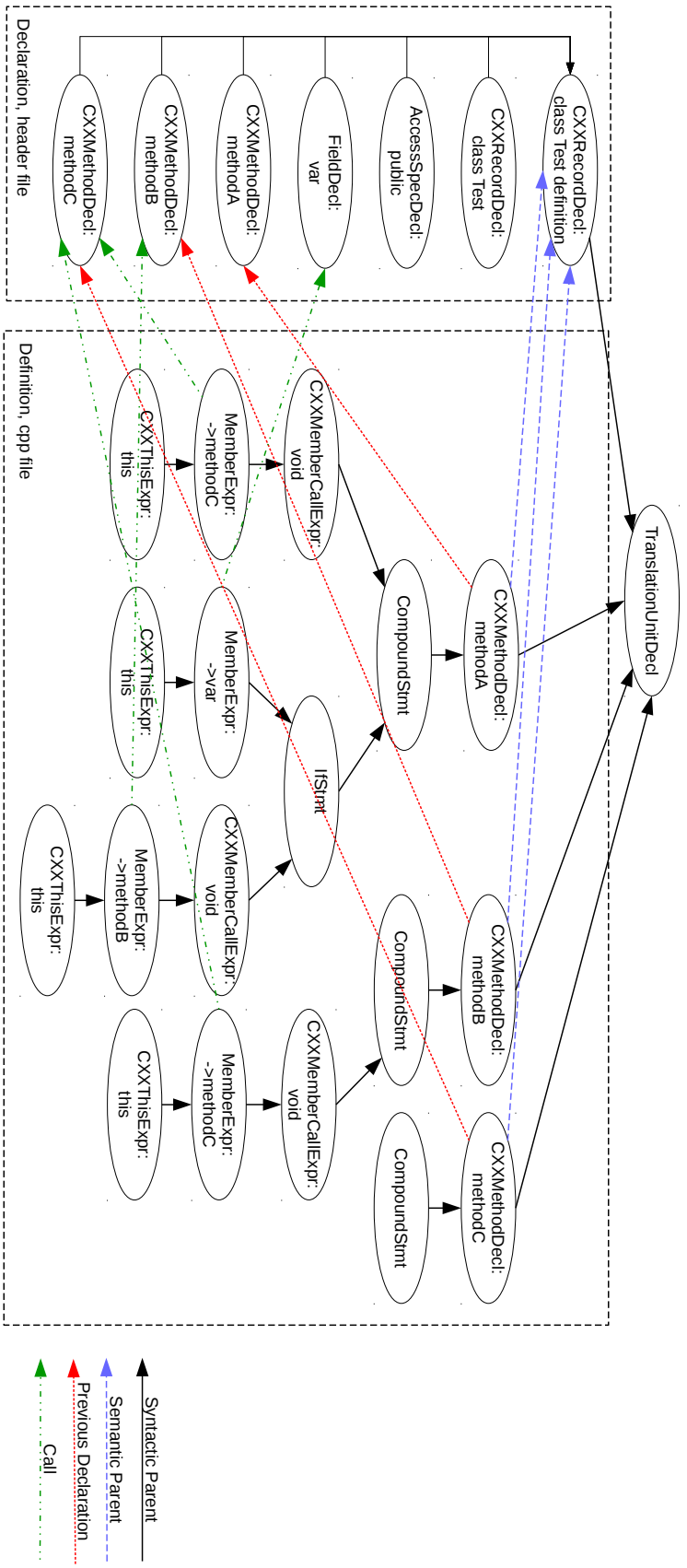


Figure 12.3.: Example of an abstract syntax tree from Clang with semantic references

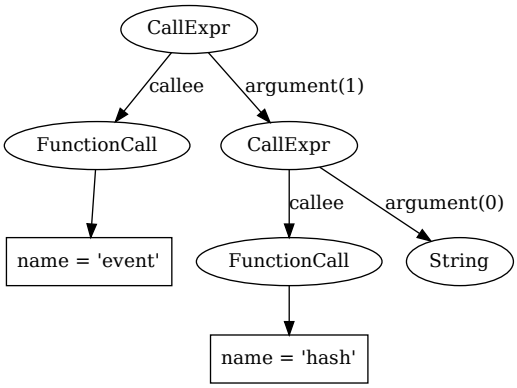


Figure 12.4.: The pattern within the abstract syntax tree to match events in the source code

arguments of the `hash` call. Finally, we save the event's complete information — the string, its hash, and the parameter's data types — into a SQLite database containing the information of the extracted model. In the next run, the model parser uses the resulting SQLite database, which contains all events of the source code, to extend the engineering model.

12.3.3 Extraction with Traversal (Example: Function Relations)

If we want to execute code on each node type with no further restrictions, we can use Clang's `RecursiveASTVisitor`. The visitor provides methods for each node type in the AST. Users inherit from the `RecursiveASTVisitor` class and override those methods they want to process. Extracting with the visitor class is what the analyzer does for extracting information about functions (and methods) in the source code. Figure 12.5 shows the class diagram for extracting function information.

For each `FunctionDecl`, the analyzer creates an entry with the function name and the location in the source code in the extended model. In addition, for each `CXXMethodDecl`, the source analyzer stores all the overridden methods into the extended model. Last but not least, it stores every function call into the extended model.

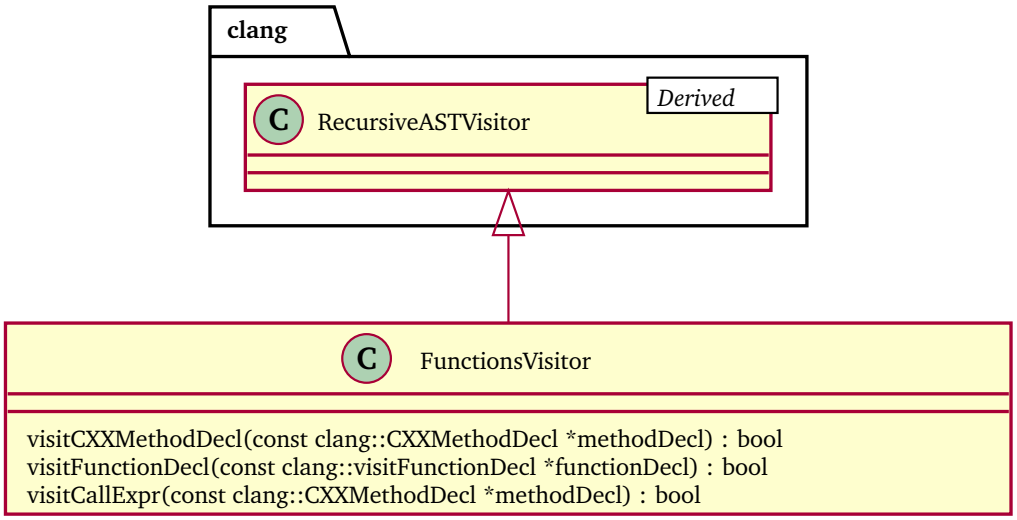


Figure 12.5.: Class diagram of the abstract syntax tree visitor for extracting function information

13 Model Verification

This chapter shows that we can perform various verification steps based on the configuration files and the model.

13.1 Simple Configuration Verifications

Every tool that works with information from the model, uses the config parser library libcorfu-config-parser. Theoretically, libcorfu-config-parser is a model transformator, from the model that is manually written by the user in yaml files into a working memory (WM) representation.

$$\mathcal{C}_{YAML} \xrightarrow{libcorfu-config-parser} \mathcal{C}_{WM} \tag{13.1}$$

Both differ in the type of representation (YAML files and binary in RAM) and the content. The parsed model $\mathcal{C}_{WM}$ builds the model for easy usage in tools, e.g., it introduces additional pointers to access elements quickly. When parsing YAML files, the parser goes through a series of verification steps: syntax, reference, ID, and asynchronous telecommands check.

Syntax Check For parsing YAML files, the libcorfu-config-parser uses yamlcpp[1], a popular YAML parsing library for C++. The yamlcpp library already accomplishes the syntax check. If there are syntax errors, it throws exceptions, which describe the problem. Hence, we did not have to implement syntax checking on our own.

Reference Check In the configuration files, some elements reference other elements, such as topics and applications. When parsing those, the config parser places them into maps with the elements' names as key. If an element references another one, the config parser retrieves it from the map. If a referenced object is not available in the appropriate map, there is an error, and, therefore, the parser throws an exception.

[1] https://github.com/jbeder/yaml-cpp

ID Check There are different entities in the model that contain an ID, e.g., applications and nodes. Indeed, IDs have to be unique. In an extra validation step, the parser iterates through the lists of those entities and checks their IDs for uniqueness. The IDs of telecommands and telemetry have to be unique on the application level. That means that several apps can have telecommands with the same ID, but telecommands within the same application might not use the same ID.

The implemented uniqueness check for IDs creates temporary maps with IDs as keys and a flag of whether it is already assigned or not. When iterating the list IDs, the algorithm checks the flag's status. If there is already a positive entry, it throws an exception. In total, this algorithm has a run-time complexity of $\mathcal{O}(n)$.

Asynchronous Telecommand Check A telecommand handling thread must be present if there is at least one asynchronous telecommand in the configuration file. Otherwise, there is no thread to process telecommands asynchronously. The other way is just a warning; a configuration that defines a thread to handle telecommands asynchronously without having asynchronous telecommands is useless.

13.2 Scheduling Analysis

The engineering model for the onboard software already contains thread and their timing information. That allows Corfu to examine scheduling properties already at the design phase.

13.2.1 Formal Verification

Thread configurations come with scheduling information like priority or timing properties for periodic threads. Hence, this information is available early and, therefore, can be investigated early. In this section, we only regard scheduling on single processor, i.e. no multi-core or multi-processor scheduling. When using Rodos in multi-processor systems, there is an own instance for every processing core, which work and schedule independently.

Basic Thread Formalization

Before we start to analyze the scheduling, we first introduce the basic notation, which we take from [7]. Table 13.1 lists the primary symbols, which we use in this chapter, and figure 13.1 depicts them on a short example of scheduling. In this example, there are two threads, τ_1 and τ_2, scheduled preemptively with fixed priorities. The second one has a higher priority than the second one. In addition, thread τ_2 has a quite big period and, therefore, is executed only once in the example schedule. Due to the greater priority, the second execution of τ_1 is suspended and finished after executing τ_2. As a consequence, the response time $f_{1,2} > f_{1,1}$. However, the execution time always stays the same, $C_1 = C_1' + C_1''$. The worst case response time of thread τ_1 is $R_1 = \max\{f_{1,1}, f_{1,2}\} = f_{1,2}$.

For periodic threads, the relase time is always an absolute time, which does not depend on other threads. Their values are already kown at compile-time.

$$r_i = \{r_{i,0} + jT_i | j \in \mathbb{N}\} \tag{13.2}$$

Based on the first release time $r_{i,0}$, we can derive the consecutive release times of threads. The value of the first release is available from the engineering model of the onboard software.

Generally, scheduling runs infinitely. However, when examining periodic thread scheduling, it suffices to use the major cycle. The major cycle contains all relative thread release combinations. The configuration of the major cycle runs iteratively.

Symbol	Meaning
τ_i	The i-th thread
$\tau_{i,j}$	The j-th instance of the i-th thread
T_i	The period of thread τ_i
D_i	The relative deadline of thread τ_i
C_i	The worst-case execution time of thread τ_i
R_i	The worst-case response time of thread τ_i
$r_{i,j}$	The release time of $\tau_{i,j}$
$f_{i,j}$	The response time of $\tau_{i,j}$
$d_{i,j}$	The absolute deadline of $\tau_{i,j}$

Table 13.1.: Notation for real-time scheduling algorithms and analysis methods (adapted from [7])

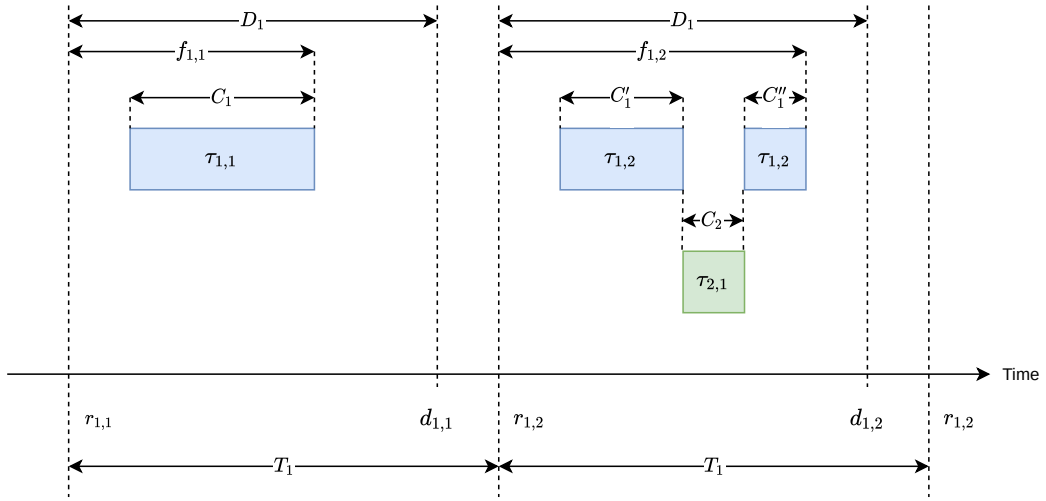


Figure 13.1.: Notation for Real-Time Scheduling Algorithms and Analysis Methods (adapted from [7])

The length of the major cycle is defined as the least common multiple (lcm) of all thread periods. Therefore, for a given node $n \in N$, the major cycle is

$$\hat{c}_n = \text{lcm}\{P_i | i \in Th_n\} \tag{13.3}$$

Utilization

Let A_n be the applications on the node $n \in N$ and Th_a be the periodic threads in an application $a \in A$, then we can define the set of threads the node n as

$$Th_n = \bigcup_{a \in A_n} Th_a \tag{13.4}$$

Let us assume context switching takes a constant amount of time S. Then, we can define the utilization of all periodic threads of a node n as

$$u_n := \sum_{i \in Th_n} \frac{C_t + S}{T_i} \tag{13.5}$$

(cf. [96])

As soon as Corfu detects a utilization of

$$u_n > 1 \tag{13.6}$$

for a processor, it can immediately report an overloaded thread configuration.

Overlapping Periodic Threads

Periodic threads are defined by two timing properties: the first execution time and the period. Threads influence other threads running on the same node. For example, several threads can be active at the same time. However, only one can run at a time; consequently, all other threads have to wait for other ones to finish. We define the set of simultaneously active periodic threads on a node $n \in N$ at a given time t as

$$AT_t^n := \{i \in Th_n | r_{i,j} < t < r_{i,j} + f_{i,j}\} \tag{13.7}$$

The priorities of simultaneously active threads define the order in which the scheduler will execute them. If at least two threads share the highest priority, the scheduler applies round-robin scheduling. Therefore, all threads will have an increased response time compared to a scheduling configuration with no simultaneously active threads. If a thread has the highest priority solely, this will be the only thread without increased response time unless the scheduler does not release a thread with a higher priority at the thread's active time. In any case, at least $|AP_t^n| - 1$ threads will have an increased response time compared to a scheduling configuration, which has no simultaneously active threads. However, the processor utilization stays almost[2] the same.

In order to minimize the response times of all threads, the scheduling should have minimal overlappings of thread executions. $\hat{O}_n$ is the maximum number of active thread overlapping over the major cycle:

$$\hat{O}_n := \max\{AT_t^n | 0 \leq t < \hat{c}_n\} \tag{13.8}$$

It is desirable to minimize the number of simultaneously active threads in order to keep response times small:

$$\min \hat{O}_n \tag{13.9}$$

Corfu calculates the overlapping values and gives engineers a report about overlapping threads. This allows engineers to adjust the timing properties in order to achieve a scheduling with minimal overlapping of thread executions.

[2]The costs of context switch may be higher than consecutive running.

13.2.2 Timing Observations

We extend the software to measure timing properties in testing environments. With the recorded information, we can see whether the software complies with the given values in the configuration file or whether the source code, software structure, or values must be adjusted.

Timing Observation of Periodic Threads

For periodic thread, the model defines the values for the first execution and the period; the generated code directly passes them to the operating system. Hence, the operating system manages to start the execution periodically. Therefore, we do not have to observe these two timing parameters in the software.

However, the framework does not pass the values of maximum execution durations to the operating system. The maximum execution durations represent estimated requirements. We use these values for the scheduling analysis. Therefore, we have to check whether the threads exceed those timings at run-time. To achieve this, we extended `corfu::PeriodicThread` (see section 8.4.3) and `RODOS::StaticThread` to record the timing behavior.

In `RODOS::StaticThread`, we add the variable `executionTime` for adding up the execution time of a thread. Every time the scheduler continues executing the thread, it saves the activation time. Whenever the thread gets suspended, it adds the time interval to `executionTime`.

In `corfu::PeriodicThread`, every time an iteration starts to execute, it resets the `executionTime` variable to zero. After a thread iteration finishes the execution, the rest of the time is added to `executionTime`, which results in the total execution time of an iteration. The verification compares the resulting value with the maximum value given in the model. If it exceeds the maximum value, the framework increments an exceedance counter variable and saves the maximum execution time.

Timing Observation of Aperiodic Threads

Aperiodic threads contain utilization values in the configuration. Also, the framework does not pass this value to the operating system and, therefore, has no direct influence on the scheduling. Like for the periodic thread, we measure the utilization in the software and record it.

To achieve this, we have extended `RODOS::StaticThread` to register the executed time before and after switching the context. The framework stores three utilization values into the thread class: one for the utilization over one minute, one over five minutes, and one over 15 minutes. If one value exceeds the utilization value from the configuration file, the framework increments an exceedance counter. The next incrementation happens after one minute at the earliest.

13.3 Stack Usage Analysis of Threads

Corfu's analysis tools extract different information from the software's code. The information includes stack usage of individual functions, function calls, and method overriding. With this information, we can estimate the stack usage of threads. Algorithm 2 shows the pseudo-code of our approach that exploits information from both the engineering and extracted models. Outgoing from a given method, the algorithm recursively traverses through the call graph via depth search.

One requirement for safety-critical software is not having recursive function calls, neither directly nor indirectly. This restriction to the code ensures that our algorithm finishes eventually. Another requirement for safety-critical software facilitates estimating stack usage: function pointers are not allowed in the code. Determining which function is actually called by static analysis is hard or even not possible. However, the code uses virtual methods, which technically are function pointers. For example, applications use virtual methods to call the actual user code. The algorithm determines the stack usage for all overriding methods and takes the largest one. At least for Coru, this is not a problem because there is usually only one user method that overrides the virtual one in the generated class.

Topics are a special case. For the communication between applications, the onboard software uses the `Topic` class of Rodos. Topics contain a list of subscribers, which the software builds up at run-time. Therefore, it is not easy to determine by static analysis which subscriber methods the software will invoke. Luckily, we have this information available in the engineering model. There, engineers define which applications subscribe to the topics. Therefore, when the traversal comes across a topic publication, the algorithm looks up the subscribers from the engineering model and continues the traversal only in the defined subscribers.

The figure 13.2 shows an exemplary graph of information. There, we have two subscribers (blue) in total, not only for the used topic. With the information from the extended model, we see that only the second subscriber on the right subscribes

Algorithm 2 Stack usage determination

```
 1: function MAXSTACKUSAGE(method)
 2:     myStackUsage ← stackUsage(method)
 3:     maxCalled ← myStackUsage + MAXCALLEDSTACKUSAGE(method)
 4:     maxOverridden ← MAXOVERRIDDENSTACKUSAGE(method)
 5:     return max(maxCalled, maxOverridden)
 6: end function

 7: function MAXCALLEDSTACKUSAGE(method)
 8:     maxStackUsage ← 0
 9:     for all calledMethod ∈ callingMethods(startMethod) do
10:         calledStackUsage ← MAXSTACKUSAGE(calledMethod)
11:         if maxStackUsage < calledStackUsage then
12:             maxStackUsage ← calledStackUsage
13:         end if
14:     end for
15:     return maxStackUsage
16: end function

17: function MAXPUBLISHEDSTACKUSAGE(method)
18:     if method is TopicInterface::publish then
19:         return MAXPUBLISHEDSTACKUSAGE(method)
20:     else
21:         return MAXOVERRIDDENSTACKUSAGETOPIC(method)
22:     end if
23: end function

24: function MAXPUBLISHEDSTACKUSAGE(method)
25:     maxStackUsage ← 0
26:     for all topic ∈ publications(method) do
27:         for all subscriberMethod ∈ subscribers(topic) do
28:             subscriberStackUsage MAXSTACKUSAGE(subscriberMethod)
29:             if maxStackUsage < subscriberStackUsage then
30:                 maxStackUsage ← subscriberStackUsage
31:             end if
32:         end for
33:     end for
34:     return maxStackUsage + stackUsage(TopicInterface::publish)
35: end function

36: function MAXOVERRIDDENSTACKUSAGE(method)
37:     maxStackUsage ← 0
38:     for all overridingMethod ∈ overridingMethods(startMethods) do
39:         overriderStackUsage ← MAXSTACKUSAGE(overridingMethod)
40:         if maxStackUsage < overriderStackUsage then
41:             maxStackUsage ← overriderStackUsage
42:         end if
43:     end for
44:     return maxStackUsage
45: end function
```

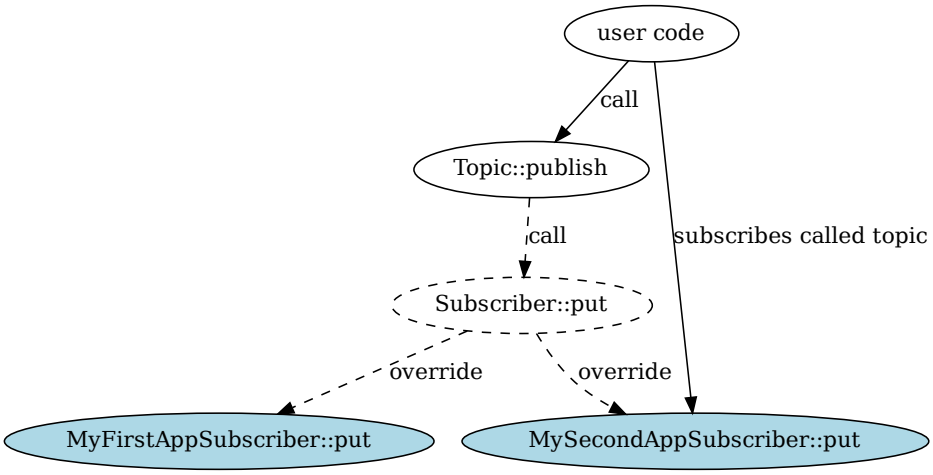


Figure 13.2.: Example of information about topic publications

the used topic. Therefore, it suffices to examine only this subscriber. Considering the model's subscription information improves the output quality because it limits the exploration space for topic subscriptions.

After all, Corfu's stack analysis overestimates the stack usages. As mentioned before, Corfu takes the maximum stack usage of all overriding methods unless it is the topic publish method. In addition, there might be compiler optimization which leads to different stack usages within a function. Corfu always takes the maximum value and assumes that the next call occurs on top of the maximum stack usage. However, the code might call a function on a lower stack level. We are convinced that the stack analysis can be further improved (also with extended feedback).

14 Automatic Testing

Automatic testing is a very fundamental part of creating robust and dependable software. Therefore, we tried our best to cover all parts of Corfu to achieve a high coverage rate with unit tests. Indeed, this does cover not only the onboard part but also the generating and ground parts. Projects that use can rely on a tested framework. However, there will be a code that is specific for each satellite mission. Developers manually write this code, and, therefore, our existing automatic tests do not cover it. Therefore, it is the developer's responsibility to test their code. Fortunately, Corfu comes with some features that support testing user code. There are two types of tests for which Corfu supports writing tests: unit and integration tests, which we present in this chapter.

14.1 Unit Tests

In contrast to other tests (e.g., integration tests), each unit test covers only a tiny part of the software. The idea is to have a simple test to cover code on a low level, isolated from other code[50]. Later, in the integration test, more extensive parts of the code are tested. Integration tests test the combination of already (unit) altogether.

In the onboard software, users provide their code in subclasses of a generated class. The generated class integrates Rodos features, e.g., Rodos threads, which the scheduler immediately executes. Having different threads running in parallel makes unit testing very hard. Therefore, we have to avoid running threads when executing unit tests.

Corfu's approach is to generate different (base) classes than for the onboard software. As stated earlier, we only want to test the user code. Therefore, having a different superclass does nothing change in the user code. On the contrary, it even simplifies testing by providing methods for checking the application output. The user code does not only have the return value as an output; there are several actions that the code can actively trigger: emitting telemetry, publishing topic messages, anomalies, or events. The regular onboard code transmits those values to topics. As we generate

test-specific classes, we can omit the transmission part and cache the messages locally instead. The unit test code can check the content of the messages after calling the method to test.

For the user-defined (sub)class, the generated base class provides several features:

- Member variables such as references to topics for publication,
- Methods, e.g., for reporting anomalies,
- Invoking of (callback) user code methods such as calling thread code or handling telecommands.

The generated testing (base) classes have to provide the same interface towards the user code. The functionality, however, does not have to be implemented in the base class because the test code invokes the user methods manually.

The regular compilation target for the onboard software includes the generated code for onboard software. For unit testing, users do not have to change the user code at all. Instead, Corfu applies another compilation configuration for the tests. Instead of including the generated code of the onboard software, the compilation process includes the generated code for unit tests. The name of the generated test class is the same as the one from the onboard software. Therefore, the user-defined class still inherits the same-named class; however, the user class inherits a different base class implementation when compiling the test executable.

If we compare figure 8.6, which contains the classes for applications in the onboard software, and figure 14.1, we can see that the class at the bottom does not change, only the classes at the top. The top class does not subscribe to a topic anymore because, for unit tests, we do not need telecommand distribution. Instead, the test code directly calls the handler methods to test them. The class for the standard telemetry is still there because this is part of the accessible interface towards the user class (at the bottom).

Listing 14.1 shows an example unit test. It contains testing code for handling `ThreadIsAlive` topic messages. Its task is to save incoming timeouts into a map within the watchdog application. The test accomplishes three steps: It inserts a timeout value, checks whether the application has inserted the correct value into the map, and checks that this is the only element in the map.

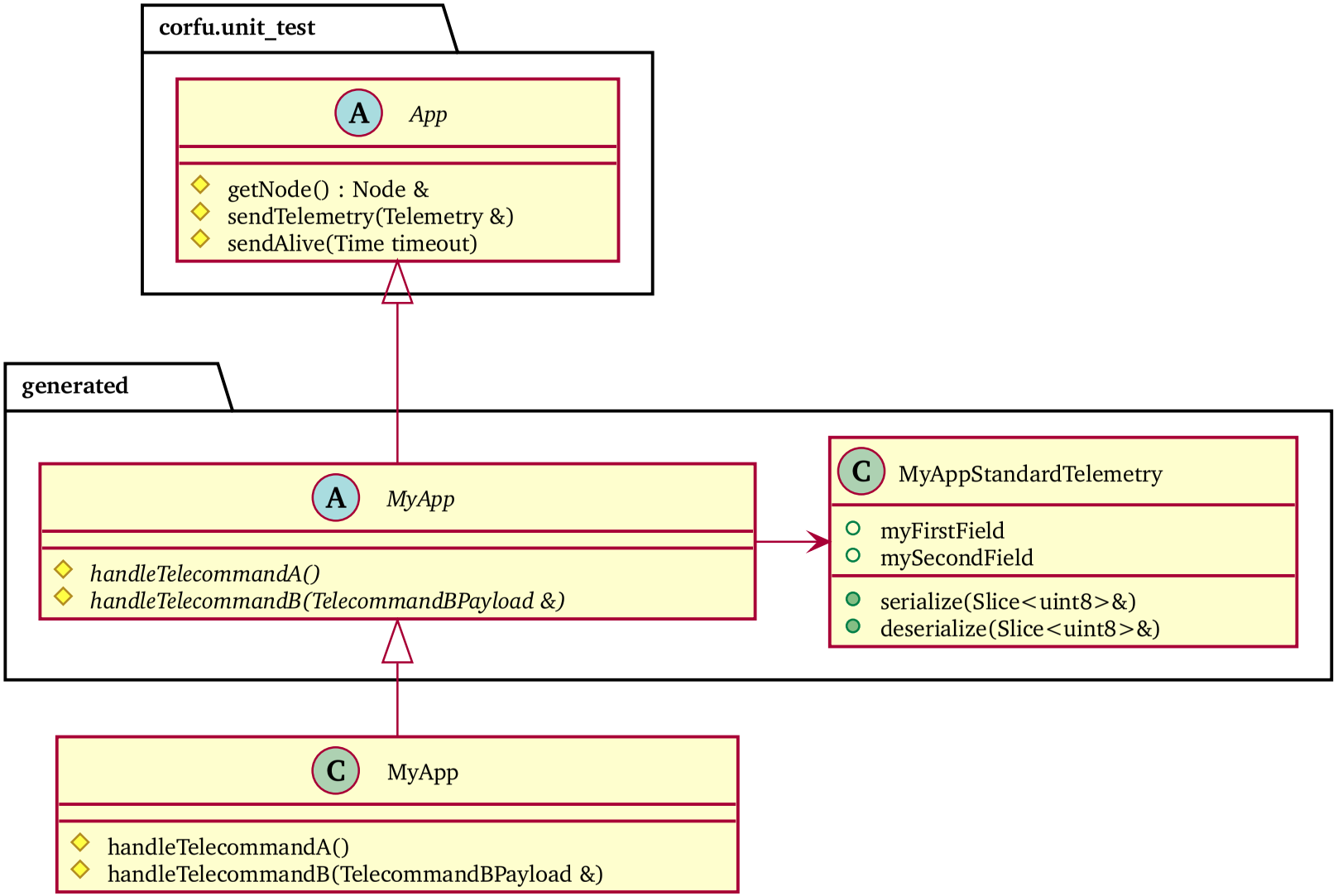


Figure 14.1.: Class hierarchy for unit testing of applications

```
TEST(WatchdogTest, handleThreadIsAliveTopic) {
  constexpr uint32_t THREAD_ID = 1337;
  const int64_t firstThreadtimeout = NOW() + 1_s;

  Watchdog watchdog;

  // insert timeout for the first thread
  EXPECT_TRUE(watchdog.handleThreadIsAliveTopic({THREAD_ID,
     firstThreadtimeout}));

  Result<int64_t> getResult = watchdog.threadTimeoutMap.lock
     ().get(THREAD_ID);
  ASSERT_TRUE(getResult.isOk());
  EXPECT_EQ(timeout, getResult.val);


  Result<int64_t> getResult = watchdog.threadTimeoutMap.lock
     ().size();
  ASSERT_TRUE(getResult.isOk());
  EXPECT_EQ(1, getResult.val);
}
```

Listing 14.1: Example unit test for updating timeout values in the watchdog application

14.2 Integration Tests

In contrast to unit tests, which only test the user code, integration tests use the entire onboard software, which includes the operating system, Corfu, the nodes, and the applications. The interface of the testing code towards the onboard software is the external interface of the applications, namely telecommands, telemetry, and events. Figure 14.2 shows the concept. The framework buffers all the data the test code receives from the onboard software. The test code checks the buffer for the data and compares it with the expected values.

Corfu provides an interface for writing integration tests, which bases on the ground library. This interface provides functions to send telecommands with required parameter values. The listing 14.2 shows an exemplary integration test.

```
sendTelecommand(
  Telecommand{nodeId, appId, telecommandId, ParameterList{
    Parameter{"parameterA", 123},
    Parameter{"parameterB", 1.23}
```

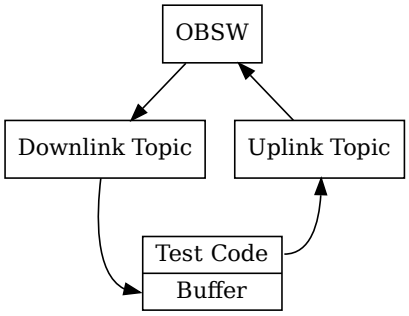


Figure 14.2.: Schema of integration tests

```
5     }}
6
7     const auto telemetry
8                   = telemetryCache.getNext<MyTelemetry>(3_s);
9     ASSERT_EQ(456, telemetry.resultA);
10  );
```

Listing 14.2: Example integration test that sends a telecommand and receives a telemetry

For checking the result, the testing framework caches every response that it receives from the onboard software, e.g., every telemetry. In the test code, users can check for the responses, e.g. for specific telemetry types. All the methods for retrieving a response have a timeout parameter to avoid blocking the automatic tests if responses are absent. In the example above, the method waits a maximum of three seconds.

Integration tests can run in a distributed way. That means the test code runs on a desktop computer or a CI server and the onboard software runs on the target platform. For communication, they use the ground/space interface. The good side effect of this is: if users do not find a way to trigger a specific behavior (code), they know that one or more telecommands are missing in the configuration and implementation.

Part V

Evaluation

15 Case Study: Log Event System

The presented events approach (see sections 8.4.5 and 12.3.2) enables sending event messages and any numbers of parameters, e.g. as telemetry. When invoking the `event` method, the software passes the following information:

1. the severity
2. the ID of the triggering application,
3. the message string,
4. the time of the event triggering, and
5. parameter values.

The message string does never change; it is a constant string literal in the source code. Every time the code triggers the same event, the message stays the same. Therefore, it is unnecessary to transmit them repeatedly. Storing and transmitting constant values is what we avoid with our approach in order to achieve better performance.

The macros `EVENT` and `EVENT0` pass the message string to the `hash` function. We declared the `hash` function as being `constexpr`[90]. Whenever there is a `constexpr` function, the compiler might execute it already at compile-time under particular requirements. For example, all the parameter types and the return type must be simple data types (i.e., scalar, reference, an array of literal type, or an appropriate class type). Therefore, the compiler calculates the hash value for the passed string already at compile-time. Thus, the software does not call the `hash` function at run-time; instead, the compiler replaces its invocation with the hash value. As a result, the string does not end up in the binary; instead, a smaller hash value does. Replacing the string with the hash value saves memory onboard and capacity on the transmission path. Especially the last advantage can be significant for small satellites, which have only a little or expensive bandwidth. In sum, with our approach, the software transmits only the following information:

1. the severity,

2. the ID of the triggering application,

3. the hash value of the message string,

4. the time of the event triggering, and

5. parameter values.

In contrast to a classical implementation without the extracted model with information from the source code, we do not need to transfer the string every time the software triggers an event. However, the compiler generates more code than the classical approach because of the template functions that serialize the events parameters.

15.1 A Classical Implementation of Event Messaging as Reference

To compare the performance of our approach, we present a comparable implementation that does not use the extracted model. There are different ways for classical implementation. For example, we could keep the template functions for `sendEvent` and `serialize`. This approach, however, would not avoid the extra template functions. In addition, it would require storing the message strings in the binary file.

Incorporating the parameters into the event message with `sprintf` is a candidate with better performance. This approach of classical event messaging does not require any additional functions for serializing parameters because `sprintf` handles this. If `sprintf` is not already available in the binary file, we must consider these additional bytes.

The listing 15.1 shows how developers can realize such a classical implementation. This function provides variadic parameters, which the software evaluates at run-time. The function passes the variadic parameters to `sprintf` and saves the message string into the event structure. Finally, it publishes the event to a topic, just as in our approach.

```
void sendEvent(Severity severity, uint8_t appId, const char*
    msg...) {
  StringEvent event{ severity, appId };
  va_list     args;
  va_start(args, msg);
  sprintf(event.string, msg, args);
  stringEventTopic.publish(event);
}
```

Listing 15.1: A classical implementation for sending log event messages

15.2 Comparison of the Bandwith Usage

Both approaches require the parameters severity, application ID, and trigger time because those parameters carry relevant information. Thus, we define c to be the number of bytes that these fields require.

$$c = |severity| + |appId| + |time| \tag{15.1}$$

This number is fixed already at compile-time.

The difference between our approach and the classical implementation is that our approach only transmits a hash value that the compiler calculates from the message text. In contrast, the classic approach transmits the full message text. Consequently, we calculate the required data length as follows.

$$L_{new} = c + |parameters| + |hash| \tag{15.2}$$

$$L_{classical} = c + |stringified(parameters)| + |message| \tag{15.3}$$

While our approach serializes the parameters in their binary form, the classical approach converts them into a human-readable format (stringified parameters). Thus,

$$L_{classical} > L_{new} \tag{15.4}$$

$$|stringified(parameters)| + |message| > |parameters| + |hash| \tag{15.5}$$

must be true that our implementation is more efficient than the classical approach regarding transmission data size. In our implementation, we used `uint32` for the $hash$ value. Therefore, if the message is bigger than four bytes, our approach uses

less memory for the transmission. In the usual case, text messages contain more than four bytes.

Depending on the parameter value, the human-readable format requires less memory if it contains fewer digits in readable format than bytes in binary format. However, we expect this to be not very common.

15.3 Comparison of Binary Memory Usage

Applying our approach saves bandwidth on the transmission part, and it also influences the memory footprint of the compiled binary file. The influence manifests itself in two areas: the string literals in the binary file and the generated code size. In our approach, no string message ends up in the compiled program; instead, the compiler replaces the messages with a hash value. Therefore, our approach saves the same amount of memory here as on the transmission path.

We assume that the classical approach reuses the function `sprintf`, already available in the binary file. The size of the `sprintf`s code has to be added to the calculation if the function is not already available in the binary file. In Rodos, for example, `sprintf` requires 392 bytes in the *.text* section on x86.

15.3.1 Determining Code Size of Our Approach

Our approach requires for each event a generated `sendEvent` functions and zero or more `serialize` functions — depending on the number of parameters. However, this does not mean that the compiler freshly generates every function for each event. If several events require identical function signatures, they are reused, i.e., the compiler generates them only once. Thus, the required code size depends on the actual content used in the events.

Even if it is possible to pass nested structures as parameters to the event, we assume developers use only basic data types, i.e., signed and unsigned integers with the sizes of 8, 16, 32, and 64 bits, as well as float, double, and bool. To assess the impact of the two template functions, `sendEvent` and `serialize`to the code size in our approach, we have measured the functions code size for different parameters. Since an infinite number of parameter type combinations exist, we have limited our measurements to homogeneous parameter combinations, i.e., to function instances that only take one

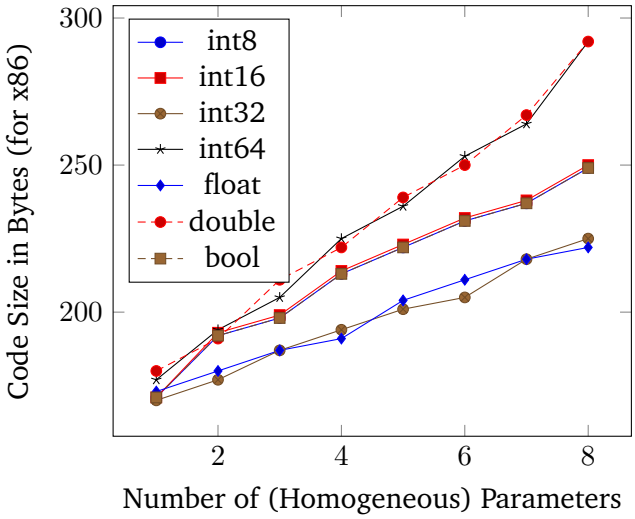


Figure 15.1.: code size of `sendEvent` functions with homogeneous parameter types

type of data. The code size of other functions, which take heterogeneous parameters, lies between the values for homogeneous parameters.

Figure 15.1 and figure 15.2 show the memory that `sendEvent` resp. `serialize` functions take up. We have compiled the code from figure 8.4 with the g++ compiler (version 9.3.0) and enabled optimization (`-O2`) for the x86 instruction set architecture. For measuring, we have instantiated each of the template functions for one to eight parameters with the same data type. We have done this for seven different data types: integer with 8, 16, 32, and 64 bits as well as `float`, `double` and `bool`.

The values for `sendEvent` in figure 15.1 show a linear increase of the size in three different slopes. For the parameter types double and int64, which require 64 bits, the function size increases with a slope of about 15.5 bytes per parameter. The code sizes with int8, int16, and bool parameters lie between the others, with a slope of about 10.5 bytes per parameter. The boolean data type requires 1 byte, just as int8. All three parameter types lead to the same slope of about 7.5 bytes per parameter. The data types that make `sendEvent` take up the least amount of memory are int32 and float, which both require 4 bytes because those values fit into the registers of x86, which are 32 bits wide. Therefore, they do not require additional instructions like smaller or bigger data types.

To determine the additional memory that invoking `sendEvent` causes, we have to consider both functions `sendEvent` and `serialize`. Invoking `sendEvent` creates one

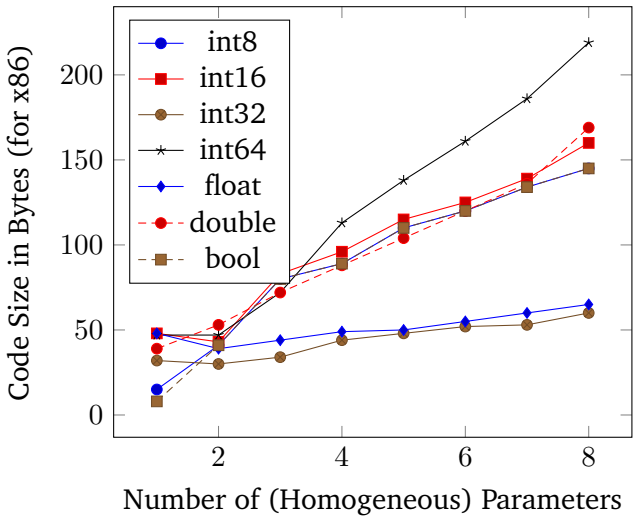


Figure 15.2.: Code size of `serialize` functions with homogeneous parameter types

instance for `sendEvent` with the required parameters and $n - 1$ `serialize` functions for n passed parameters to `sendEvent`.

If we describe the list of parameters for one event as a sequence, this is the set of parameters that the generated `serialize` functions use:

$$P_e = \{< p_1, p_2, \ldots, p_n >, < p_2, \ldots, p_n >, \ldots, < p_n >\} \tag{15.6}$$

Here, the $p_i \in P_e$ elements stand for single parameters required for an event e. Thus, the number of `serialize` functions for an event e is $|P_e|$, because the compiler creates a function for each sequence.

$$F_e^{serialize} = \{f_{serialize}(p) | p \in P_e\} \tag{15.7}$$

In this equation, $f_{serialize}(p)$ describes a `serialize` function for the given parameters p. However, the compiler must not generate all `serialize` functions for each event. If there are `serialize` instances with the same sequence of parameters (i.e., the same signature of the function), they are reused and not generated multiple times. Therefore, the final set of `serialize` functions that the compiler generates is the union of the `serialize` functions for all events.

$$F^{serialize} = \bigcup_{e \in E} F_e^{serialize} \tag{15.8}$$

Here, the set E contains all the events in the system. This step removes all multiple existing functions because sets do not contain the same element multiple times.

Thus, $|F^{serialize}|$ gives the total number of `serialize` functions that the compiler generates from the template. Based on $F^{serialize}$ we determine the actual size for the code of `serialize` functions.

$$b_{serialize} = \sum_{f \in F^{serialize}} |encode(f)| \tag{15.9}$$

The function $encode(f)$ represents what the compiler does: encoding the `serialize` function into binary code. Summing up the length of all generated `serialize` functions in the system results in the total size required for serializing all parameters of events.

For the size requirement of the `sendEvent` functions, we also sum up the number of bytes.

$$b_{sendEvent} = \sum_{f \in F_{sendEvent}} |encode(f)| \tag{15.10}$$

In our approach, the message string does not end up in the binary file; instead, the compiler replaces it with the corresponding hash value. Also here, we add up the number of bytes required to save the hash values in the binary file:

$$b_{message} = 4|M| \tag{15.11}$$

Here, the set M contains the event messages of the system. Since we use four bytes for the hash value, we multiply it with the number of messages in the system.

Finally, we determine the total sum of additional code size for our approach by adding the size values for both types of functions and the hash values:

$$b_{new} = b_{serialize} + b_{sendEvent} + b_{message} \tag{15.12}$$

The result, b_{new}, represents the number of bytes that our approach additional requires in the onboard software to send events.

15.3.2 Determining the Code Size of the Classical Approach

For the classical approach, the additional memory comprises the message strings. $\mathcal{M}$ denotes the set of message strings that the source code contains. Thus, the number of bytes for the classical approach is the sum of the length of each message string.

$$b^{classical} = \sum_{m \in \mathcal{M}} |m| \tag{15.13}$$

As mentioned earlier in this section, we assume that the binary code already contains a `sprintf` function. If this is not the case, it adds up to a few hundred bytes. For example, the implementation of `sprintf` requires 392 bytes in Rodos.

16 Comparison with Our Classical Onboard Software Implementation

The basic principle of Corfu follows the maxim of Rodos: to keep it "as simple as possible" [1]. Hence, we do not want to introduce much complexity into onboard software. Therefore, we tried to keep the generated code as simple as possible; so that the generated code is still easily reviewable. Simple generated code also means that it is very lightweight; it does not introduce unexpected things like threads that do not show up in the configuration. Hence, the overhead resource consumption of Corfu is not unnecessarily significant.

This chapter compares our classical implementation approach of onboard software, which the chair of aerospace information technology has used in other missions, with the one of Corfu. We compare different properties for the different parts: the number of logical lines of boilerplate code, which users have to write manually, the static memory consumption, and the (binary) code sizes.

16.1 Software Elements

This section has a look at the different components of applications. It gives an overview of our classical implementation of onboard software, which we have used in recent projects, and compares them with Corfu's approach.

16.1.1 Application Structure

Similar to Corfu, our classical implementation divides the software into different applications. Our classical implementation does not come with generic classes for applications. Instead, our classical approach statically instantiates everything globally in the source files. Listing 16.1 shows the components of classical applications.

[1] https://gitlab.com/rodos/rodos/-/blob/master/README.md

There is a namespace for each application, containing the application-specific classes, which we present in the following sections. In addition, an instance of `Application` is created representing the application.

```
1 namespace MyApp {
2   Application myApp("myApp", MYAPP_ID);
3
4   // specific structures and variables/instances
5
6   // classes for handling telecommands, standard telemetry,
7   // topic subscribtions, and threads
8 }
```

Listing 16.1: Example of our classical application implemenation

In Corfu, users create their own (sub)class for each application instead of having a namespace. Having a class allows users to encapsulate their data.

```
1 class MyApp : public generated::my_app::MyApp {
2   // specific structures and variables/instances
3
4   // classes for handling telecommands, standard telemetry,
5   // topic subscribtions, and threads
6 }
```

Listing 16.2: Example of Corfu's application implemenation

16.1.2 Symmetric Telecommand Handling

In our classical approach, developers have to create a custom class that handles telecommands. Listing 16.3 shows an example class for handling telecommands. In this code, there are several lines, which every application repeats. In the beginning, in lines 6 to 8, the `put` method checks whether the incoming telecommand addresses the application. If the telecommand addresses the present application, the method deserializes the parameter values into a struct.

```
1 MyAppCommandIF : public SubscriberReceiver<Telecommand> {
2 public:
3   MyAppCommandIF()
4     : SubscriberReceiver<Telecommand>(telecommands, "MyApp")
        {}
5
6   void put(Telecommand &telecommand) override {
7     if(telecommand.appId != myApp.getID()) {
```

```
8         return;
9       }
10
11      TelecommandParameter parameter;
12      parameters.deserialize(telecommand.serializedPayload);
13
14      bool validParameter = true;
15
16      executedTelecommandsCount++;
17
18      switch(telecommand.telecommandId) {
19        case TelecommandIds::MYAPP_NOP:
20          // process telecommand
21          break;
22
23        case TelecommandIds::MYAPP_TELECOMMAND_A:
24          if(parameter.parameterA >= PARAMETER_A_MIN &&
               parameter.parameterA <= PARAMETER_A_MAX) {
25            // process telecommand
26          } else {
27            validParameter = false;
28          }
29          break;
30
31        default:
32          anomalyTopic.publish(AnomalyId::BAD_TC);
33          rejectedTelecommandsCount++;
34          executedTelecommandsCount--;
35      }
36
37      if(!validParameter) {
38        anomalyTopic.publish(AnomalyId::BAD_TC_PARAM);
39        rejectedTelecommandsCount++;
40        executedTelecommandsCount--;
41      }
42    }
43  }
```

Listing 16.3: Example of our classical telecommand handling

Our classical implementation tracks the number of successfully and erroneous telecommand executions in two global variables accessed by all applications on a node: `executedTelecommandsCount` and `rejectedTelecommandsCount`. Consequently, those variables have to be incremented accordingly in every application. The

example code does this in lines 16, 33, and 34. However, having to write such lines in every application manually is error-prone. Corfu encapsulates this in base classes. The same goes for handling undefined telecommand ids in the `default` part of the `switch` construct. These lines are also almost the same for all applications — just the anomaly ID is different.

In general, Corfu's approach of handling telecommands in applications is limited to just implementing the telecommands handling code. The generic and generated application classes take care of typical work from the user's source code, includes checking telecommand addressing, invoking telecommand handling functions, taking care of telecommand execution counting. Listing 16.4 shows what users have to implement for handling the same telecommands as before. The generated code checks even the parameter range if defined in the configuration. Users can fully concentrate on implementing the actions for telecommands.

```
1 Error handleNopTelecommand() override {
2   // process telecommand
3   return NO_ERROR;
4 }
5
6 Error handleTelecommandA(TelecommandAPayload &payload)
    override {
7   // process telecommand
8   return NO_ERROR;
9 }
```

Listing 16.4: Example of telecommand handling in the user code of corfu

16.1.3 Standard Telemetry Handling

For standard telemetry, Corfu follows a different approach than our classical implementation. Our classical approach uses topic messages to request data for the standard telemetry from each application. The data type of the topic is the full standard telemetry of the node. Hence, all applications have direct access to the entire standard telemetry structure. They might damage the values of other applications, even unintended. Corfu's approach avoids this because applications only have access to their part of standard telemetry.

In the Rodos communication middleware, the communication middleware passes data as a non-const reference. Therefore, subscribers can modify the passed data. The changes are also available for the next subscribers that process the data. Our

classical implementation of standard telemetry exploits this fact. Every application subscribes the `standardTelemetryRequest` topic and places information into their fields. After all subscribing applications have done that, the housekeeper retrieves a filled packet of standard telemetry, which all applications have filled with information. Listing 16.5 shows how our classical implementation realizes real-time handling in applications.

```
class StdTMIF : public SubscriberReceiver<StandardTelemetry>
    {
 public:
  StdTMIF()
    : SubscriberReceiver<StandardTelemetry>(stdTMRequest, "
        MyApp") {}

  void put(StandardTelemetry &telemetry) override {
    // fill fields of telemetry paramter
  }
}
```

Listing 16.5: Example of our classical standard telemetry handling

In our classical implementation of standard telemetry, users have to manually implement a handling function for putting data into the node's standard telemetry structure. In most cases, the handling method copies data from some variable into the telemetry structure.

In Corfu, updating standard telemetry is decentralized. Apps have an instance of their part of standard telemetry. They update this member variable as soon as new data arrives. There is no need for intermediate variables. The values are directly stored in the structure that the housekeeper retrieves periodically.

However, some values for the standard telemetry might not be available for free; instead, some might require calculation or retrieval. It might be more efficient to update such values right before the housekeeper collects the data. Corfu supports this by providing the virtual `updateStandardTelemetry` method in the generic application base class. Users can override this method in order to update vacant data in the standard telemetry structure, as can be seen in listing 16.6.

```
void updateStandardTelemetry() override {
  // fill fields of telemetry member variable
}
```

Listing 16.6: Example of standard telemetry handling in corfu

16.1.4 Periodic Threads

In our classical implementation, users have to create subclasses for each thread they want to create manually — see listing 16.7. Users must make sure that they pass the correct configuration values for the stack size and timing parameters. Corfu generates the subclass automatically. The user does not have to care about configuring the threads accordingly. Instead, the generated code takes care of configuring the threads. Users only have to implement their code for initialization and periodic code, as can be seen in listing 16.8.

```
class MyAppThread : public StaticThread<STACK_SIZE> {
  void init() override {
    // initialization (without scheduler)
  }

  void run() override {
    // initialization (with scheduler)

    TIME_LOOP(FIRST_RUN, PERIOD) {
      // periodic code
    }
  }
} myAppThread;
```

Listing 16.7: Example of our classical thread implementation

```
Error MyThread::initializeBeforeScheduler() {
  // initialization (without scheduler)
}

Error MyThread::initializeWithScheduler() {
  // initialization (with scheduler)
}

Error MyThread::runIteration() {
  // periodic code
}
```

Listing 16.8: Example of thread user code in corfu

16.1.5 Topic Subscription

Our classical implementation does not provide any convenience code for implementing topic subscriptions. Instead, users have to do this manually by using Rodos classes. Listing 16.9 shows an exemplary classical topic subscription.

```
class MyTopicSubscriber : public SubscriberReceiver<
    MyTopicType> {
 public:
  MyTopicSubscriber()
      : SubscriberReceiver<MyTopicType>(myTopic) {}

  void put(MyTopicType &message) override {
    // handle topic message
  }
}
```

Listing 16.9: Example of our classical topic subscription

In Corfu, the generated code already handles creating receiver subclasses, and manually registering the topic is not required. Users only have to override and implement one method for handling message data from the topic. Listing 16.10 shows an exemplary implementation of a topic subscription in Corfu.

```
void handleMyTopic(MyTopicType &message) override {
    // handle topic message
}
```

Listing 16.10: Example topic subscription in corfu

16.2 Comparison of Both Implementations

This section contrasts the two implementation approaches in several aspects.

16.2.1 Logical Lines of Manual Code

One goal of Corfu is to minimize the number of lines of code that users have to write manually. Table 16.1 contains the number of logical lines users have to write in order to implement certain aspects in the source code. In sum, users only have to write more lines than our classical approach for the coarse application structure. All other elements require fewer manual lines of code. The number of lines in telecommand

	App	Telecommand Handling	Providing Real-Time Telemetry	Topic Subscriber	Periodic Thread
Classical	1	17 + 2 per TC	5	5	8 to 9
Corfu	2	2 per TC	0 to 1	1	1 to 2

Table 16.1.: Comparison of logical line of code between our classical implementation and Corfu

handling in both cases depends on the number of telecommands that the application should handle.

For handling standard telemetry, our classical implementation requires users to implement a topic subscriber for the `standardTelemetryRequest` topics. That leads to *five* logical lines of code users have to implement. In Corfu, users do not have to write additional manual code for copying values to the standard telemetry because the values are already available in the standard telemetry member variable. However, if values have to be calculated just for the standard telemetry, users have to override a method, which takes *one* logical line of code.

Our classical approach requires users to implement a subclass for each thread and pass parameters manually when implementing periodic threads. This results in *eight* to *nine* logical lines of code. For Corfu, users have to override only one or two methods; therefore, it requires only *one* or *two* logical lines of code.

16.2.2 Static Memory Usage

We define static memory usage as the amount of data that software allocates statically. Hence, it does not include memory on thread stacks. Figure 16.1 shows the usage of static memory for different software elements in the two implementation approaches: our classical one and Corfu. To determine those values, we have measured the sizes with C++'s `sizeof`. Chapter C in the appendix shows data diagrams of the different software elements. Those diagrams further divide the required static memory size into separate elements.

For the base application structure, Corfu requires less memory because the `Application` class in Rodos implements a linked list, which inflates the static memory. Corfu has the list of applications moved into the node class, where this memory emerges.

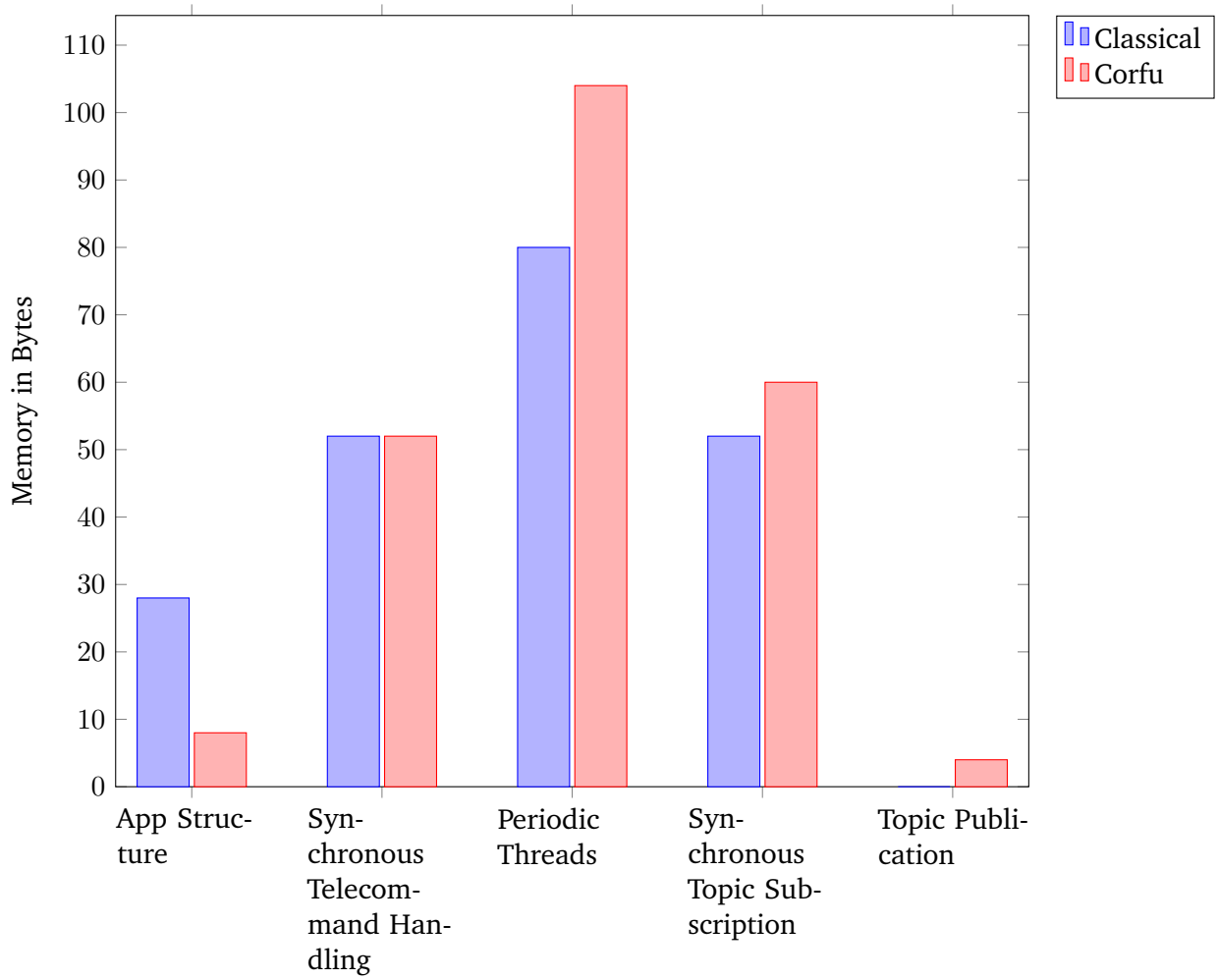


Figure 16.1.: Comparison in static memory usage of different software elements

Synchronous telecommand handling uses the same amount of memory for both implementations; all other features require more static memory for Corfu than our classical implementation.

16.2.3 Code Size

When programming software, there are usually tradeoffs between different aspects. The most known is the space-time tradeoff, which says that algorithms can be faster but require more memory, or they are more frugal with memory but require more time to finish[37]. Another tradeoff is between static (run-time) memory and code size. For example, maps can be encoded into the code by using `switch` constructs or into working memory (static and stack memory) in map objects. Therefore, the two

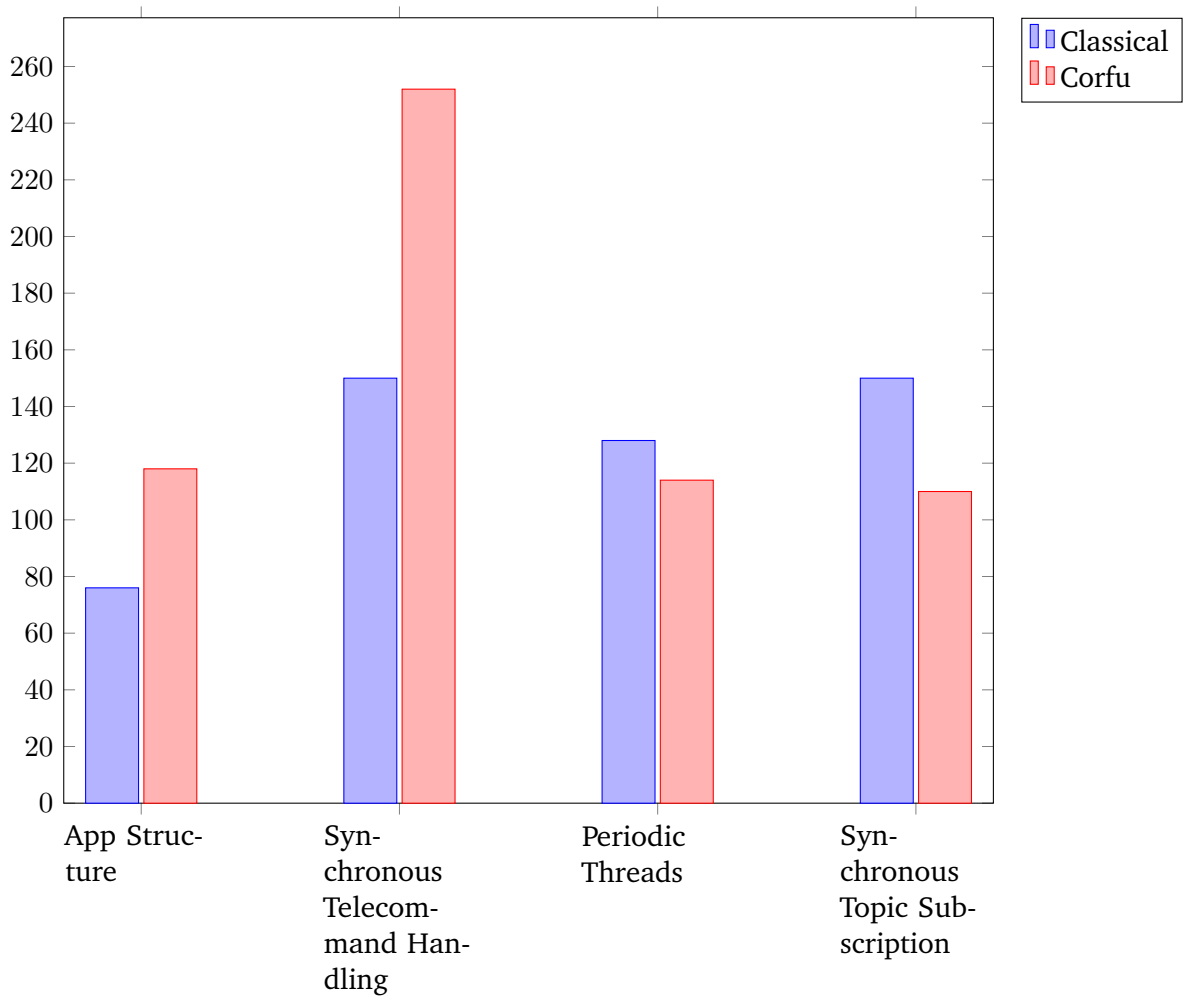


Figure 16.2.: Comparison in code size of different software elements

diagrams of static memory, 16.1 from the section before, and of code size, 16.2 can be considered together to see their tradeoff.

The diagram of the code size shows the total code size, including generic, generated, and user-written code. There, we see a split picture. While the app structure and the synchronous telecommand handling require more code in Corfu, periodic threads and synchronous topic subscriptions require less code than our classical implementation of onboard software.

17 Development Process Evaluation

Several aspects contribute to fulfilling non-functional software requirements and their metrics. Such metrics are the number of bugs in the source code and the time in development and testing. In this chapter, we have a look at Corfu regarding such different aspects.

17.1 Avoided Bugs

When designing Corfu, avoid programming errors was one of the focuses we had. To achieve this, Corfu introduces several software constructs, making it nearly impossible to introduce specific bugs in the source code, which this section shows.

17.1.1 Forgetting to Use Semaphores

Semaphores are a common source for run-time faults. The most delicate issue is that such errors often occur only at specific interactions between threads, making them particularly hard to debug. There are some common errors when working with semaphores and resources For example, not locking corresponding semaphores when accessing resources is such an error. Extending the source code might introduce another type of error: when developers introduce a new condition branch, they might forget to unlock semaphores in new branches.

Implementing and using guard semaphores in combination with hidden data can avoid such errors. Listing 17.1 shows such an implementation. The idea is this: instead of having open access to the protected resource, it is encapsulated into the class `ThreadSafeData`. The same class also contains a semaphore for protecting the shared resource. Whenever users intend to access the resource, they have to request an instance of `SemaphoreGuard` from the `ThreadSafeData`. The instance of `SemaphoreGuard` is created on the stack and automatically locks the semaphore in the constructor. It allows developers to access the variable while the corresponding

semaphore is locked. As soon as the scope is left, the `SemaphoreGuard` instance is destroyed, which revokes the variable's access and automatically leaves the semaphore.

```
template<typename Type>
class SemaphoreGuard {
  Semaphore &semaphore;
  Type &data;

  public:
  SemaphoreGuard(Type &data, Semaphore &sema)
  : data(data), semaphore(sema) {
    this->semaphore.enter();
  }

  ~SemaphoreGuard() { semaphore.leave(); }
};

template<typename Type>
class ThreadSafeData {
  Type data;
  Semaphore semaphore;

  public:
  ThreadSafeData() = default;

  ThreadSafeData(const Type &data) : data(data) {}

  SemaphoreGuard<Type> lock() {
    return SemaphoreGuard<Type>(data, semaphore);
  }
};
```

Listing 17.1: Implementation of a semaphore guard

Figure 17.2 shows the usage of `ThreadSafeData`. This mechanism provides a good hint that developers must access the values in a thread-safe way. However, be aware that it could still be misused if a reference of the protected variable is stored externally.

```
ThreadSafeData<int> safeData;
{
  SemaphoreGuard guard = safeData.lock();
  if(something) {
    guard.data = 123;
```

```
6     } else {
7       guard.data = 456;
8     }
9   }
```

Listing 17.2: Usage of ThreadSafeData

It is possible to implement a more efficient variant of `ThreadSafeData`. There would be two `access` methods, one returning a const semaphore guard and one returning a mutable semaphore guard. Hence, users can select whether they need to modify the data or whether they intend just to read it. As long as there is no writing access, parallel reading accesses do not block each other.

17.1.2 Copy/Paste Errors in the Applications

Chapter 16 presents how our classical onboard software implementation lookes like. It shows that there is boilerplate code which is the same for all applications. For example, such boilerplate code covers error reporting for telecommands. Having the same or similar code at different places in the source code has several drawbacks:

- Changes to the code have to be manually transfered to all copies, which is error-prone.
- It might be inteded Developers might not notice small discrepancies in the copied code. As a consequence, they might expect the code to behave differently than it actually does. Developers might implement their code in the wrong knowledge and therefore introduce bugs.

Corfu encapsulates all generic application code either in the generated superclasses. Basically, it outsources manual copying to the code generator.

17.1.3 Accidentally Set Other Application's Standard Telemetry

Our classical approach (see chapter 16) has only one structure for standard telemetry per node. For collecting standard telemetry values, it passes one variable to all the applications in the node. Hence, applications can also access and overwrite fields of other applications.

Corfu, instead, has separate standard telemetry structures for each application. Therefore, it is not possible to (accidentally) write to other application's fields.

See section 8.4.2 for information about how Corfu implements standard telemetry handling.

17.1.4 Detecting Stack Excess only at Run-Time

Threads execute user code. Therefore, the actual stack usage depends on the user code implementation. Corfu introduces a static analysis of the user code. In this way, it is able to already calculate stack usage of the threads already at compile-time and compare it with the configuration from the model. Corfu's stack analysis examines all execution paths, event those that will execute rarely, which might be hard to find by testing. Even if the calculated stack usage is not exact, it is able to report early warnings for developers allowing them to modify the configuration in the model.

17.1.5 Forgetting Checking for Anomalies

In Corfu's engineering model engineers can define which anomalies telecommand and thread handlers must report. Due to static analysis, it can be checked, whether the source code actually emits those anomalies or not. If this is not the case, we can assume that the code does not check for the anomaly.

17.1.6 Unintended Modification of Constant Variables

There are often variables, which are assigned a value only once. After that, the software does not modify their values anymore; they are just read. This intention is made clear by using the keyword `const` as often as possible. Additionally, it lets the compiler complain about misusing the variable.

Another similar keyword is `constexpr`, which already has a known value at compile-time. Developers should always clarify this characteristic by using `constexpr` and not by preprocessor (`#define`).

There are cases where only a superclass should modify a member variable; subclasses should only read the variable. Making a variable modifiable only by a superclass can be enforced by hiding the variable from the subclass and providing an accessing method that references the value. Listing 17.3 shows an example. The node variable can always be called by the subclasses as long as the superclass makes sure that it is

no invalid (null) pointer. The good thing: the superclass can be sure that no other class in the inheritance hierarchy changes the pointer.

```
class App {
 private:
  Node *node = &myNode;

 protected:
  Node &getNode() { return *node; }
};

class MyApp : public App {
  void myFunc() {
    node = nullptr;         // ERROR
    getNode.someMethod();   // OK
  }
}
```

Listing 17.3: Example of protecting a variable of a base class

According to the requirement to make variables constant when possible leads to constant object whenever possible. For constant objects, users can only call constant methods. Thus, making `const` is not limited to variables; also, methods should always be declared as constant when possible.

```
class Result {
  ErrorCode error = NO_ERROR;

  public:
  bool isOk() const {
    return error == NO_ERROR;
  }

  ErrorCode getError() { return error; }
};

const Result result = someFunction();
if(result.isOk()) {   // OK
  result.getError();  // ERROR
}
```

Listing 17.4: Example for using constant methods

Listing 17.4 shows an example. The returned `result` from the function should be constant because it represents the result of the function called and should not be

changed afterward. However, if one forgets to set a method to be constant, this is not possible, as one can see in the example above. The method `getError` can simply be attributed with `const` without any problem. Having such methods as `const` makes the `Result` class usable in a constant environment.

17.2 Potential New Bugs

The previous section showed that Corfu avoids several potential programming errors that users can introduce. However, as most approaches, Corfu also comes with some drawbacks regarding bugs. Even if Corfu's non-onboard libraries and tools are not safety-critical, their output is. It is essential that the config parser and the generator provide correct generated code. By providing unit tests with a high code coverage we try to mitigate this issue, but it is still there.

17.3 The InnoCube Cubesat Project

Innocube is the first satellite, which uses Corfu as the framework for its onboard and ground software[38]. It is a joint project of the Technische Universität Berlin and the Julius-Maximilians-Universität Würzburg.

17.3.1 Introduction

The InnoCube satellite demonstrates two innovative technologies in space: SKITH and Wall#E. SKITH stands for "Skip The Harness" and is a technology for wireless communication between the computing nodes on a satellite. SKITH aims to replace the wired bus between the computing nodes with a wireless connection between all participants. A wireless connection between the components overcomes integrating a bus system on the backplane and avoids faulty connections between the computing nodes. The satellite has six different types of computing nodes. That also means six different types of onboard software configuration running on the satellite. However, different node types reuse apps, making this satellite a perfect candidate for a software framework like Corfu. Thus, in this project, we have apps that run on several (or even all) nodes and some that run only on one type of node, e.g., scientific software on payload computers. The satellite launch is scheduled for 2023.

17.3.2 Development Experiences

When developing the software for InnoCube, we had some experiences regarding different aspects.

Reduced Development Time As mentioned before, Innocube is the first real satellite mission relying on Corfu. Developing Innocube's onboard software showed that configuring and implementing applications is faster and less erroneous than the former approaches. Developers could directly transform some requirements into the software configuration. The generator directly creates the software structure from it without manual coding.

In previous satellite projects, developers have created new apps by copying a source code template and modifying it accordingly. On the one side, this is error-prone due to the manual work; on the other side, later changes to the structure of all applications require manual modification of all applications. With Corfu, Innocube developers did not have to copy any source code manually. Instead, Corfu encapsulates the basic structure of apps in generic and generated code. Writing only the configuration file and not copying and manually modifying a template avoids inadvertent mistakes and saves time implementing the apps.

Better Maintenance In addition, developers could easily apply changes to the general structure of all apps. They can modify the generic code or template files for the generator. Corfu ensures that re-generation does not modify the user code.

Corfu's configuration files for onboard software allow defining software components concisely and clearly. Users can be confident that such software components are implemented the same in the generated code. If they had to implement such components by hand, it could be possible that they behave differently in different applications.

Another helpful tool when developing the onboard software with Corfu was the generating of documentation. They also help developers interactively navigating across the software configuration. When source code is generated, the generator also creates linked HTML documentation files containing tables and diagrams that give a good overview of the software's structure.

Better Testability In our classical approach, the source code directly used operating system resources, such as threads. This makes unit testing difficult. Usually, unit tests only cover single functions or methods. However, if the user code is tangled with the operating system, starting the full operating system is necessary. Having the operating system running introduces behavior that testers might not desire. For example, there should not run other threads which might get in the way.

Corfu compiles the user code against different code for testing, which uncouples the user code from the operating system. In this way, it is possible to test only single functions or methods. This allows test-driven development for the user code.

Better Resource Analysis In contrast to our classical approach, Corfu encapsulates applications into its own classes. On the one hand, this clarifies responsibilities within the software; on the other hand, this eases analyzing the software. For example, we can determine applications' memory usage by applying `sizeof` on them.

Simplified Ground Software Integration Corfu comes with a library for interfacing the onboard software. It shares source code with the onboard software for the telecommand and telemetry interface. When developers create their specific ground software, they can program against Corfu's ground library to communicate with the onboard software.

People at our chair have developed a virtual control room[87]. For the communication link to the onboard software, they have incorporated Corfu's ground library. The integration worked seamlessly. By using the ground library, the virtual control room can interact with each Corfu-based satellite. The same applies to a web interface for satellite communication, which we have implemented.

In addition to the ground library, Corfu comes with ground software, which dynamically builds a graphical user interface for commanding the onboard software. It is immediately available when starting the development, which makes it perfect for rapid development. Developers do not have to implement any code to get started, which also reduces the development time.

Part VI

Conclusions

18 Summary

Corfu is a framework for satellite software, not only for the onboard part but also for the ground. Developing software with Corfu follows an iterative model-driven approach. The basis of the process is the engineering model. Engineers formally describe the basic structure of the onboard software in configuration files, which build the engineering model. In the first step, Corfu verifies the engineering model. Not only syntactically and semantically but also on a higher level such as scheduling.

Based on the model, Corfu generates a software scaffold, which follows an application-centric approach. Software images onboard consist of a list of applications connected through communication channels called topics. Corfu's generic and generated code covers topic communication, as well as telecommand and telemetry handling. All users have to do is inheriting from a generated class and implement the behavior in overridden methods. For each app, the generator creates an abstract class with pure virtual methods. Those methods are callback functions, e.g., for handling telecommands or executing code in threads.

However, from the model, one can not foresee the software implementation by users. As an innovation compared to other frameworks, Corfu introduces feedback from the user code back to the model. In this way, we extend the engineering model with information about functions/methods, their invocations, their stack usage, and information about events and telemetry emission. Indeed, it would be possible to add further information extraction for additional use cases. We extract the information in two ways: assembly and source code analysis. The assembly analysis collects information about the stack usage of functions and methods. Corfu's source code analysis relies on Clang to identify patterns in the abstract syntax tree containing the extracted information.

On the one side, we use the gathered information to accomplished additional verification steps, e.g., checking if stack usages exceed stack sizes of threads. On the other side, we use the gathered information to improve the performance of onboard software. In the event logging use case, we have shown how using the extended model at run-time can reduce binary sizes and bandwidth usage towards the ground can.

19 Future Work

The presented approach of an extended model-driven development still has much potential, which we have not fully implemented yet. We have several ideas for exploiting static analysis and the model-driven approach for small satellite software.

Introducing hardware devices into the engineering model The software usually relies on device drivers for commanding hardware devices such as sensors, actuators, and busses. Such drivers configure the hardware and provide a programming interface for the applications. Moving the hardware configuration to the engineering model allows engineers to provide configuration parameters formally. On the one hand, the generated code could instantiate the device drivers with the correct configuration. On the other hand, defining devices already in the engineering model allows more extensive testing. For executing stacks, the framework could instantiate dummy device drivers instead. The test code could simulate sensors to check the behavior of the software from external stimulation. For actuators, the test code can check the outputs that the software has set.

Automatic Stack Size Adjustment If Corfu's stack analysis detects an overrun, the user will receive a report at compile-time in the current implementation. The analysis process only extends the engineering model and does not modify it. It is conceivable to extend the process also to adjust specific parameters of the engineering model. For the stack size, this means that the stack analysis tool might auto-generate the stack size parameters of the threads.

Automatic Thread Timing Adjustment Like the idea in the previous paragraph, it involves overriding configuration values from the engineering model. Our current implementation of scheduling analysis only detects overlapping periodic threads based on the given timing parameters from the model. Engineers define the period and execution times, which cannot be changed automatically. They must be considered fixed. The first execution time, however, has only minor importance. Usually, engineers can choose it freely as long as the period is respected. Therefore, we could extend the scheduling analysis to determine suitable combinations of first execution

times to achieve overlappings of thread activations. The analysis tool can do this for each node separately.

Checking Static Memory Usage at Compile-Time Thanks to the application encapsulation in the source code, it should be possible to gain detailed information about the applications' memory usage and their components. Having information about the memory consumptions of individual software parts can be helpful for developers when investigating issues. In the next step, we could also introduce new parameters in the engineering model, allowing engineers to fill their estimated memory usages. Corfu can then compare the estimated with the actual values and report warnings early in the development phase.

Static Access Rights Checking When separating the software into different applications, engineers know which application is responsible for which devices or tasks. However, the developers who implement the applications might misunderstand the informal knowledge of the engineer. After introducing devices into the engineering model (see the previous paragraph), we could also introduce a simple form of rights checking. By static analysis, the code analyzer can extract information about which app accesses which device. Based on the method calls, it could also distinguish between reading and writing access. With both information, the implementation details from the extracted model, and the intended device access from the engineering model, the framework can check whether apps are not correctly accessing devices.

Part VII

Appendix

Communication Middleware

A

This appendix chapter presents different communication middleware implementations that are freely available and it describes different implementation aspects of communication middleware.

A.1 Existing Middleware

Several concepts for communication middleware for both intra-node and inter-node communication are available. Some of them are also freely available under open-source conditions. This section presents a selection of available communication middleware.

A.1.1 Core Flight System's Software Bus

The Software Bus is part of the software framework for space applications named core Flight System (cFS)[104, 105]. NASA's Jet Propulsion Laboratory (JPL) developed the cFS, which is available under open source licenses[1]. The cFS uses C as the programming language; hence, it provides a procedural programming interface. The Software Bus uses the same packet format for data exchange between all onboard applications. Currently, it comes only with the implementation of the CCSDS Space Packet. However, the Software Bus is extensible to support other packet formats by extending the code. The Software bus labels each packet with an ID, which it uses to identify the packet content type and the route to its destinations. Following the publish-subscribe principle, onboard applications can register for those packet type IDs to receive all messages containing that ID.

The Software Bus itself only provides message passing within one computing node. There is a different application for communication across multiple partitions or computing nodes: the Software Bus Network. This application acts as a gateway between the local Software Bus and some communication hardware, e.g., a network connector, to exchange messages remotely.

[1] https://github.com/nasa/cFS

A.1.2 Rodos Middleware

Rodos[34, 68] is a full software stack for embedded systems with focus on aerospace applications, which is available under Apache License[2]. It comes with a communication middleware for exchanging messages between applications, which follows the publish-subscribe principle. The software instantiates topic objects, connecting the data producers (publishers) with the data consumers (subscribers). A topic is equipped with an ID and has a fixed data type defined at compile time. A gateway service converts the messages for topics into network/bus messages and sends them via hardware communication channels to the other node(s) for communication with other computing nodes. The gateway service reads the message on the destination node and delivers the message to the local topic subscribers.

The gateway's job is to forward local messages to remote nodes that are reachable directly and vice versa. To enable communication between nodes that are not directly connected, Rodos provides a separate router class, which forwards messages between several gateways.

A.1.3 Outpost's Simple Message Passing Channel

Outpost is a software platform for spacecrafts developed by the German Aerospace Center (DLR). It comes with a communication library called Simple Message Passing Channel (SMPC). Like Rodos, communication channels are created by instancing topic objects, which define the type of transmitted data at compile time. The Outpost framework consists of two main parts, the core, and the satellite repository. Just the core is available under open source conditions[3], while the satellite part is not publicly available. According to publications, the closed-source part of Outpost contains additional libraries, a stack for CCSDS and PUS protocols[25].

A.1.4 NanoSat MO Framework's MAL

The Consultative Committee for Space Data Systems (CCSDS) published a concept for a service-oriented structure of software in both parts, the space, and the ground segment. They call this standard CCSDS Mission Operation (MO). A fundamental part of the standard is the message abstraction layer (MAL), which enables application services to define their communication interfaces. By applying the MAL,

[2] https://gitlab.com/rodos/rodos
[3] https://github.com/DLR-RY/outpost-core

developers define the communication endpoints of service providers on the one side. On the other side, service consumers can use those communication endpoints for data exchange. The endpoints can select from a variety of communication strategies. It ranges from simple message passing, with and without acknowledgment and response, to the publish-subscribe principle.

The NanoSat MO Framework (NMF) is a reference implementation of the CCSDS Mission Operations (CCSDS MO) standard created by ESA and the Graz University of Technology[23]. It is available under ESA Public License[4]. They have written NMF in Java; therefore, it requires an operating system that provides a Java Virtual Machine to run. OPS-Sat, one of the satellites that use NMF, runs a "lightweight version of Linux" [22] as the operating system. The base concept of NMF is to provide an application-like structure of the onboard software. It lets the developer describe the individual applications as services. Other (consumer) services access those services.

A.1.5 MAL C API

Another implementation of the MAL of the CCSDS MO standard (see the section about NanoSat MO Framework above) is the MAL C API. CNES developed it and made it available under MIT license[5]. The MAL C API comes with support for two underlying transport protocols: ZeroMQ and TCP connections. The project provides several APIs as adaption points for extensions. For example, the MAL C API allows developers to implement custom transport protocols or adapters. In addition, it encapsulates the encoding/decoding of messages with a well-documented API to enable custom encoding and decoding mechanisms for the MAL.

A.1.6 Cordet

The Cordet framework results from the Component-Oriented Development Techniques project[21]. It is a service-oriented framework for space software written in C[6]. A key implementation concept of Cordet is the hierarchical structure of components. Every element in the framework inherits directly or indirectly from a generic `Component`, representing a base state machine. Each inheriting component extends this state machine by introducing new states and transitions. The diverse

[4]https://github.com/esa/nanosat-mo-framework
[5]https://github.com/CNES/ccsdsmo-malc
[6]https://github.com/pnp-software/cordetfw

types of components embody elements with unequal responsibilities. A message is a descendant of the base `Component` the same as a component serving as a communication adapter that forwards messages. The selected programming language C, however, does not support inheritance directly. Consequently, the source code does not formally represent the component structure and, thus, there is no inherent type safety for the inheritance at compile time.

Cordet uses the PUS standard to communicate between the space and the ground segment and the communication within the spacecraft. However, they designed it flexible enough to support different types of exchange messages. Each message sent, either a command or a report traverses different state machine states that check whether the content is valid. The software passes a valid message to one of the transmitting components, forwarding the message to the destination node. At the receiving node, the software traverses the message component's state machine again, checking for the validness of the message before passing it to the user application's input component. Cordet supports some high-level routing, i.e., when a message with another destination address is received, the component forwards the packet to the next node or the destination node if it is directly reachable. Hence, it does not require any routing in the communication layers below its communication middleware.

A.1.7 KubOS

KubOS[76] is a project that provides a complete platform for developing and running space software[7]. KubOS comes with a customized Linux distribution as the operating system. On top of that, KubOS provides several libraries that the user applications can use with different functionality, i.a. communication between applications. The applications define a service interface that other applications can use to communicate with each other. KubOS uses web technology for the communication interface: they define the service interfaces with GraphQL, and the data connection uses HTTP as the communication protocol between the applications.

Investigating the implementation details of the communication middleware implementations presented in section A.1, we extract specific approaches and technical concepts. Those aspects are may be familiar to multiple implementations or unique to one framework. This section examines them and compares them with each other.

[7] https://github.com/kubos/kubos

A.2 Implementation Aspects of Communication Middleware

There are different aspcects of communication middleware that influence the timing behavior, memory footprint, and programming approach. This sections presents and compares some implementation details.

A.2.1 Memory Management for Messages

The software has to store the messages between applications in the memory for processing. That applies to both local and remote communication. In the set of communication middleware frameworks used in space, we found three types of memory management, which we present in this section. The approach affects the reliability and resources part of the middleware.

Dynamic Memory

The two middleware implementations NMF and MAL C, use dynamic memory as the programming language provides it. For every new message, they allocate memory (e.g., by calling `calloc` or `new`) and — if the programming language does not come with a garbage collector — the memory is manually freed after processing the message. However, developers have to free the memory eventually. Otherwise, there is a risk for memory leaks, which is the reason why some programming guidelines for safety-critical software, e.g., the power of ten[42], propose not to use dynamic memory. If an application allocates much space, for example, by allocating big or many message buffers, it might impede other applications from allocating new message buffers. As a result, other applications might not be able to communicate with other applications or with the ground segment. Thus, dynamic memory should be very error-prone and requires disciplined programming.

Message Pool

In this variant of memory management for message packets, the framework allocates memory used for message allocation at run-time at the initialization stage. Whenever an application or the framework generates a message, it allocates a message buffer from the message pool. When the destination application consumes the message,

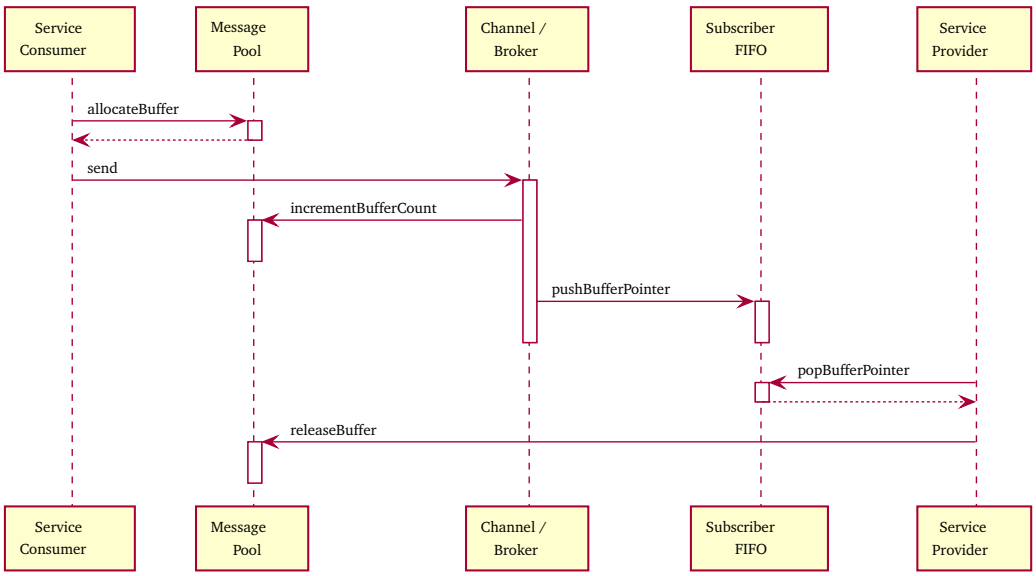


Figure A.1.: Sequence of services using a message pool for messages

or the framework sends the message to the network, they free the memory buffer. The message pool has a fixed size already at compile time. Therefore, the message allocation does not affect other parts of the software because the maximal size of memory is reserved.

Figure A.1 shows how two applications — sender and receiver — use the message pool to allocate and destroy a suitable message buffer. The sending application allocates a message buffer for the message with a suitable size. Similar to heap memory, the allocate function marks allocated memory buffers as "in use"; it reuses (re-allocates) the message buffer only after the software has freed the message buffer. During the entire process of data passing, the message buffer stays "in use."

This concept even works in communication middleware implementations that support more than one receiver. In this case, software should use reference counting, for example, shared pointers. Calling `freeBuffer` on the message pool does not always immediately free the message buffer. Instead, it decrements an associated usage counter variable on each call. If the counter falls to zero, it actually frees the message buffer, i.e., made available for other messages. To make this work, every time a pointer is created or copied, the software increases the associated counter.

The message pool approach does not influence other parts of the memory and, thus, does not affect other applications. However, it suffers the most disadvantages of dynamic memory. If the software allocates and frees message buffers of different sizes, the memory pool might become fragmented. However, by applying appropriate

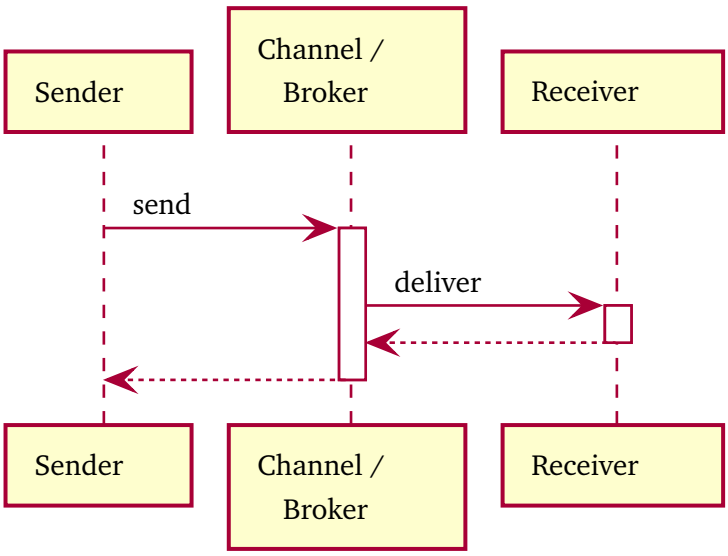


Figure A.2.: Sequence of services using the stack for messages

mechanisms, the level of fragmentation could be reduced. For example, keeping the size of the allocated message buffers fixed ensures that there are no free message buffers with unmatched lengths. However, keeping the message lengths the same is only possible for specific situations, e.g., when all allocated messages have identical sizes. Otherwise, always allocating the maximal size a message can take might lead to a waste of memory.

Using the Stack

In contrast to the two approaches described earlier, the stack usage method does not use dynamic memory. Instead, senders store the messages directly on the stack for passing them to the senders.

After the sender calls the function on the broker, the broker calls a receiver function to pass the message, as figure A.2 shows. Depending on the implementation, the message is either passed by value or by reference. There is no need to count any references because the calling functions pass the messages (e.g., reference to memory) on the stack as a function parameter; after the function call is left, the software reduces the stack, and the copy of the value or the is gone. The purpose of the called function on the receiver's side is to handle or save the required information from the message to a local memory location within the receiving application.

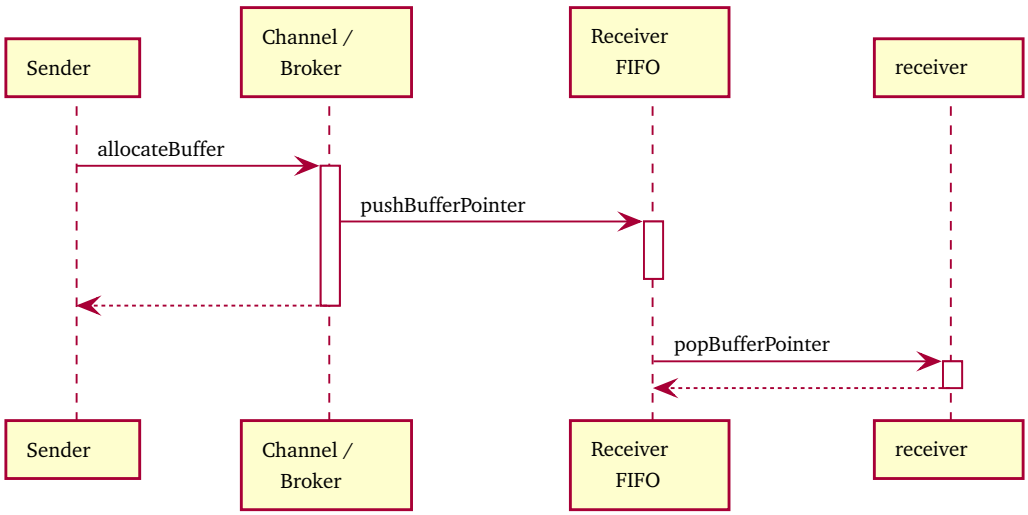


Figure A.3.: Sequence of services using an intermediate FIFO

This approach requires that each thread is equipped with enough stack memory to handle the transmitted messages. As threads do not share their stacks, this leads to more reserved memory.

Buffer for Each Channel

Another approach, which is one option in Rodos, is to store the received messages directly into a FIFO that is part of the receiver application. The difference to the message pool technique is that there is no pool for allocating message buffers. Instead, the FIFO already statically contains the required memory, which is big enough to hold a given number of messages. Every time an application sends a message, it copies the message's content into the FIFOs of each receiver application; see figure A.3. Consequently, every receiver has to provide a FIFO buffer with a fixed size, which should be big enough to handle the expected amount of received messages. In contrast to the message pool, the FIFOs do not share their memory; thus, the FIFO approach, in total, requires more reserved memory.

In contrast to the stack usage approach, the receiver is responsible for reading the message data from the FIFO. In order to avoid congestion in delivery, the receiving function must read the FIFO in a frequency that is equal or higher than the frequency of filling the FIFO. Another approach — if there is no thread to invoke the receiving function periodically — is to use a dedicated thread for the receiving function, which sleeps until new data arrives at the FIFO. The thread might be woken up directly by the FIFO.

A.2.2 Execution Context

The various approaches of memory management outlined in section A.2.1 lead to another classification of the different implementations: the execution contexts of the receiver function. The selected approach affects the real-time properties of the software because different execution contexts use different priority parameters.

Execute Code of the Receiver in the Sender's Context

The sequence diagram in figure A.2 shows that the "stack usage" memory management passes messages between applications via function calls. As a result, the sending application's execution context (thread) runs the code responsible for consuming the message at the receiving application. Consequently, there is less predictability of 1. the execution time of a sending thread, and 2. the required stack size of a sending thread. The thread of the sending application will execute the (unknown) code of other applications. Disciplined coding of receiving functions is sensible to reduce uncertainties, e.g., writing only short code that does not need much space on the stack. The receiver application cannot control when the communication middleware invokes the receiving code and which thread will execute it. Thus, synchronization is mandatory here if the subscription code accesses the resources of the receiving applications.

Execute Code of the Receiver in the Receiver's Context

Another approach is to invoke the receiver code in a thread that belongs to the receiving application. A consequence of not using function calls for message passing is that the stack is not usable. Instead, the communication mechanism must buffer the message data between sending and receiving times, as the figure A.3 shows. Here, a thread of the receiving application has to process the data of the messages. If applications introduce new threads for data receptions, it increases the complexity of scheduling analysis. This approach also increases the memory demands because of the extra stack space required for the extra threads.

A.2.3 Type Safety of Message Data

Communication middlewares can apply different degrees of type-safety. This section presents different approaches from existing middlewares, which impact the robustness of the software.

Strongly Typed Message Data

This implementation detail exploits the type mechanism of the programming language (e.g., C or C++). For example, the communication middleware implementations of Rodos and Outpost provide a template class for channels (named topics). This template parameter defines the type of the message data at compile time. Thus, sending messages whose type does not fit is impossible because passing a wrong data type leads to compilation errors.

Run-time Type Information

In contrast to the strongly typed message type, NMF uses run-time type information to detect message types. The most significant difference is that the message types are unknown to the compiler; thus, message typing errors cannot be detected and reported when compiling. However, run-time type information is often disabled to save memory and run-time, which inhibits this approach.

Binary Message Data

Other implementations, e.g., the message passing of cFS, do not exploit the typing mechanism of the programming language for passing messages. Instead, they store and forward plain binary data to the receiving applications, which is formatted. The common format comes with meta-information for identifying the message's data type. An advantage of this approach is that only one routing infrastructure suffices because no extra classes or objects are necessary for distinct message types.

A.2.4 Local Routing

The communication middleware frameworks organize local routing by different approaches. This section presents the decentralized and central approaches.

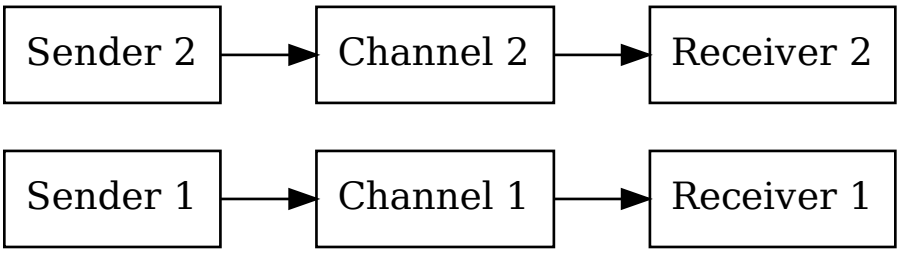


Figure A.4.: Structure of decentral local routing

Decentralized

When using the decentralized approach, there are distinct channel objects for each message type. Those channels forward messages to the destinations. For example, in implementing Outpost's and Rodos' communication middleware, the user instantiates an object for each message channel. In order to send messages of a particular type, the software uses the corresponding channel object to send the data to the receivers. All senders and receivers access the same channel object. The channel objects are independent, i.e., they do not have any interconnection. Therefore, even if one channel ceases to function, the other channels keep working.

Centralized

Another approach is handling the distribution of messages in a centralized way. Centralized message distribution is suitable if a framework uses the same binary packet format for all virtual communication channels between applications. The packet format contains all the necessary information to apply routing to the receiver applications. Here, we found two addressing methods: point-to-point routing and channel routing.

Point to Point Routing In point-to-point routing (P2P), the (source) applications address destination applications when sending messages. Therefore, the packet format must contain the address of the receiving application, which the sending application has to know. The routing mechanism forwards the message according to

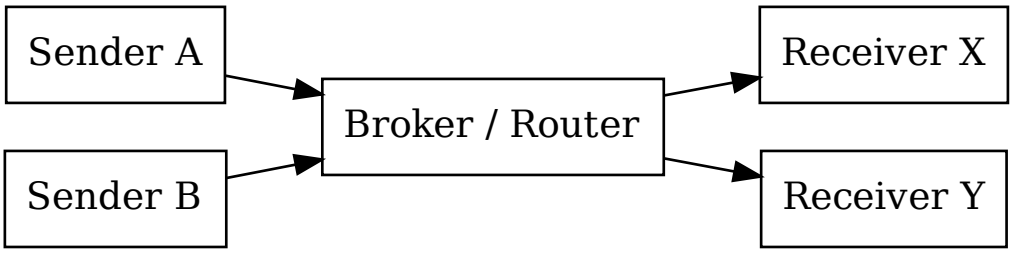


Figure A.5.: Structure of central local routing

the destination address, either 1. to the application if it is local, or 2. to the remote node that runs the application. Dependent on the complexity of the spacecraft architecture, this concept might require elaborated routing algorithms.

Channel Routing In this routing approach, the message metadata does not contain a specific receiver. Instead, senders tag the messages with channel identifiers, which allows the routing algorithm to determine the receivers. There are many ways to implement the routing algorithm. One of them is the principle of publish-subscribe, where applications can register at the routing module to subscribe for specific message channel identifiers.

A.2.5 Remote Communication

The previous section presented routing techniques for local communication, i.e., information exchange between applications running on the same node. Some investigated middleware frameworks, such as NMF, MAL C, Cordet, and KubOS, do not differentiate between local and remote communication. They use the identical mechanism for delivering messages locally and remotely. Other implementations, however, use some local routing that is not usable remotely, e.g., because they use local memory for data exchange. However, most such middleware implementations come with an extension that enables remote communication, i.e., that forwards local messages to remote nodes. One approach is to provide an application to forward messages remotely: the gateway. The gateway docks onto the local routing and converts the messages for remote delivery. Every time it receives a message with

a destination that resides on a remote node, it serializes and transmits the data via a network or a hardware bus. The software runs a gateway application on the remote node that de-serializes the received message and passes it to the local router. Rodos and cFS implement this mechanism. An exception is Outpost, which does not provide remote communication in its open-source part.

A.2.6 High-Level Routing

Remote communication only exchanges messages between nodes that share the same network or bus. If communication is across different networks, common nodes have to implement message routing. Most frameworks support different underlying communication protocols, which usually already provide different routing features. However, only a few, like NMF, MAL C, and KubOS, rely on such protocol features. Other middleware implementations, such as Rodos and Cordet, implement custom routing at a higher networking layer independent of underlying protocols. According to the metadata given in the header of the packages, the middleware decides whether the current node is a destination node. If this is not the case, the middleware forwards the message to one or more nodes. Depending on the topology of the spacecraft network, several hops might be necessary to e For this feature, Rodos comes with its custom configurable routing class. Cordet, on the other side, has no extra routing object. Instead, it immediately checks whether the destination node arriving packets is the current one in the input object. If this is not the case, Cordet forwards the packet to the appropriate output object.

A.2.7 Communication Patterns

For communication between applications, simple message passing in one direction sometimes is not sufficient. In order to ensure command executions, the communication framework should send acknowledgment and messages containing result values. The communication middleware frameworks Outpost, Rodos, and cFS do not implement responses directly. If the users desire such behavior, they have to implement it on the application level. They could implement existing standards here, such as the Packet Utilization Standard (PUS) defined by ESA. Cordet already implements the PUS protocol for on-board communication and, therefore, comes with an acknowledgment mechanism.

While Outpost, Rodos, and cFS only implement message passing via publish-subscribe, NMF and MAL C provide plenty ofă communication schemes for application interfaces:

Send This scheme sends a message to the service provider without acknowledgment.

Submit This scheme sends a message to the service provider. The receiver returns an acknowledgment message to the service consumer.

Request This scheme sends a message to the service provider. The receiver returns a response message with a payload to the service consumer.

Invoke This scheme sends a message to the service provider. The receiver immediately returns an acknowledgment message to the service consumer. After the service provider has fully processed the invocation, it sends a message containing the return value back to the service consumer.

Progress This scheme sends a message to the service provider. The receiver immediately returns an acknowledgment message to the service consumer. Periodically, the service provider sends information about the processing status to the service consumer. After the service provider has fully processed the invocation, it sends a message containing the return value back to the service consumer.

Publish-subscribe The service consumers register themselves at a broker service for a message ID. Whenever the service provider has new data, the provider sends a message to the broker service, which forwards the message to each registered service consumer. The service provider does not provide acknowledgments or responses to the service customer.

A.2.8 Communication protocol

Regarding the communication protocol for remote communication, we divide our set of communication middleware frameworks into two types: those that implement open protocol standards and those that use their custom protocols.

There are widely used protocol standards for on-board communication and communication with the ground station. The middleware frameworks of cFS (using CCSDS Space Packets), NMF, MAL C (both using CCSDS MO), and Cordet (using PUS) support such standards. KubOS applies GraphQL via HTTP for communication

between applications. Initially, these protocols are web protocols and do not origin from space agencies. Other middlewares do not follow broad standards. Instead, they come with their custom protocol for inter-app communication, for example, Rodos. Depending on the hardware and operating system, they use established protocols on the lower level, e.g., UDP, but they do not rely on them.

B Software Requirements

The software development process of Corfu followed the classical approach of defining a list of requirements; we have designed and developed Corfu based on predefined requirements. In this section, we list and describe requirements in different categories.

B.1 Requirements of Satellite Software

Corfu will run on board of satellites and, therefore, must comply with all requirements common for satellite software. This list is a very generic one; it does not commit to specific requirements of individual satellite missions. For concrete missions, it is necessary to extend the list of requirements, e.g., for hardware limitations, interfaces, and mission goals.

ID: REQ-SAT-01
Text: The software architecture shall support multiple computing nodes.
Justification: Satellites usually consist of more than just one onboard computer.
Fullfilled: Yes. Corfu's engineering model knows the concept of nodes.

ID: REQ-SAT-02
Text: Applications shall be reusable across multiple computing nodes.
Justification: Some applications should run on several computers in a satellite.
Fullfilled: Yes. Corfu's engineering model knows the concept of nodes. In addition, it comes with configuration (compile-time and run-time) parameters for applications, which improves usability.

ID: REQ-SAT-03
Text: The satellite shall process telecommands.
Justification: Satellites should be commandable.
Fullfilled: Yes. Corfu directly supports telecommands.

ID: REQ-SAT-04
Text: The satellite shall check telecommand parameters for reasonableness.

Justification: Unexepected parameter values could lead to unintended behavior.
Fullfilled: Partly. Corfu's model lets engineers define parameter ranges. However, users have to check interrelated restrictions in their code.

ID: REQ-SAT-05
Text: If a command packet is damaged, the satellite shall discard it.
Justification: Executing manipulated commands can be dangerous.
Fullfilled: Yes. Corfu handles this in its onboard library.

ID: REQ-SAT-06
Text: The satellite shall notify the operations crew about each executed telecommand or discarded.
Justification: Some commands should only be executed if the previous one succeeded.
Fullfilled: Yes. Corfu collects this information.

ID: REQ-SAT-07
Text: Telecommands shall be distributed in the satellite to reach the destination computer.
Justification: Usually, telecommands are addressed to specific onboard computers.
Fullfilled: Yes. Corfu uses a Rodos topic for distributing telecommands. Via gateways, it is possible to distribute them on board easily.

ID: REQ-SAT-08
Text: The software shall handle anomalies shall in the possible lowest level.
Justification: For example, if there is a problem at the level of software drivers, it should be recovered there.
Fullfilled: Partly. Handling anomalies at different levels is a task of the users.

ID: REQ-SAT-09
Text: The software shall propagate unrecovered anomalies to the upper level.
Justification: If the software cannot solve failures at a lower level, recovery mechanisms in a higher level might be necessary.
Fullfilled: Yes. In Corfu's model, engineers can define anomalies for reporting via topic.

ID: REQ-SAT-10
Text: The satellite shall inform the operations crew about emerging anomalies.
Justification: The operations crew should not miss any problem that occurs in the

satellite.
Fullfilled: Yes. The supplied reference implementation of the anomaly collector provides anomaly information in the standard and extended telemetry.

ID: REQ-SAT-11
Text: Satellites shall define different modes of operation.
Justification: The modes define which components are turned on/off.
Fullfilled: Not directly. Implementing the operation mode is a task of the users.

ID: REQ-SAT-12
Text: Satellites shall define a safe mode, which ensures survival.
Justification: We have to save the mission, even when serious failures occur.
Fullfilled: Not directly. Implementing the safe mode is a task of the users.

ID: REQ-SAT-13
Text: If a critical failure occurs that cannot be recovered automatically, the satellite shall automatically switch into safe mode.
Justification: Try to save our mission.
Fullfilled: Not directly. Implementing the safe mode is a task of the users.

ID: REQ-SAT-14
Text: The satellite should not automatically leave the safe mode.
Justification: The problem should be investigated by the operations team before the safe mode is left manually.
Fullfilled: Not directly. Implementing the safe mode is a task of the users.

ID: REQ-SAT-15
Text: When switching to the safe mode, the satellite shall execute a sequence of commands that set all parts of the satellite into the safe mode.
Justification: Entering the safe mode should be fully automatic.
Fullfilled: Partly. The reference implementation of the timed commander comes with telecommand lists; implementing the safe mode is a task of the users.

ID: REQ-SAT-16
Text: The sequence of commands for entering the safe mode shall be modifiable from the ground.
Justification: If the operations crew detects faults in the command sequence, they should be able to fix them.
Fullfilled: Partly. The reference implementation of the timed commander comes

with telecommand lists; implementing the safe mode is a task of the users.

ID: REQ-SAT-17
Text: The satellite shall periodically send a short overview of its components' states (standard telemetry).
Justification: The operations crew should always be aware of the satellite's current status.
Fullfilled: Yes. Corfu comes with the concept of standard telemetry in the model.

ID: REQ-SAT-18
Text: All components shall use uniform physical units (SI), internally and in interfaces.
Justification: Using unambiguous units dimishes the chance of use wrong values.
Fullfilled: Partly. It mainly depends on the user implementation. However, Corfu supports developers by providing own data types for the different SI units.

ID: REQ-GND-01
Text: If a telemetry packet is damaged, the ground software shall discard it.
Justification: Evaluating damaged telemetry data could lead to wrong conclusions.
Fullfilled: Yes. Corfu's ground library checks the integrity of telemetry data.

ID: REQ-GND-02
Text: The framework shall come with a ground software for developing purposes.
Justification: Developers can immediately start and test their code.
Fullfilled: Yes. Corfu comes with a dynamic ground software using the model.

ID: REQ-GND-03
Text: The framework shall come with a library for connecting custom ground software with the TC/TM system.
Justification: High flexible integration into existing or custom ground systems.
Fullfilled: Yes. Corfu comes with library for ground software.

ID: REQ-APP-01
Text: Applications shall be able to contain threads.
Justification: Applications should be equipped with active execution paths.
Fullfilled: Yes. Corfu's model allows users to define threads.

ID: REQ-APP-02
Text: Telecommands shall be addressed to applications.

Justification: Most applications react on application-specific telecommands.
Fullfilled: Yes. Corfu's telecommand structure addresses nodes, applications, and telecommand types.

ID: REQ-APP-03
Text: Applications should contribute some fields to the standard telemetry.
Justification: This provides an overview of application-internal states to the operations crew.
Fullfilled: Yes. In Corfu's model, applications can define a list of fields they contribute to the standard telemetry.

ID: REQ-APP-04
Text: Applications shall define a set of parameters.
Justification: Parameters influence the behavior of applications. Making applications configurable also facilitates their reusability.
Fullfilled: Yes. Corfu's model supports defining configuration parameters of applications.

ID: REQ-APP-05
Text: Configuration parameters shall be modifiable via telecommand.
Justification: The operations crew can change the behavior of applications in orbit.
Fullfilled: Yes. Corfu can generate such commands automatically.

ID: REQ-APP-06
Text: Applications shall report event messages.
Justification: Events give hints to the operations crew about what is happening in the satellite.
Fullfilled: Yes. Corfu comes with a log event system.

ID: REQ-SAT-19
Text: The satellite shall be able to execute telecommands at given time points automatically.
Justification: Most academic satellite missions do not have 24/7 contact with the satellite. Therefore, the software should provide a way to execute commands in non-contact situations.
Fullfilled: Yes. Corfu comes with a reference implementation of a timed commander.

ID: REQ-SAT-20
Text: The satellite shall be able to store a sequence of commands that it executes

consecutively at once.
Justification: This allows planing a list of actions first and execute them later at once.
Fullfilled: Yes. Corfu comes with a reference implementation of a timed commander.

B.2 Requirements of Safety-Critical Software

ID: REQ-SAFE-01
Text: Safety-critical software shall guarantee timing properties.
Justification: Due to the interaction with the physical world, reactions should happen in given time boundaries.
Fullfilled: Yes. Corfu relies on the real-time operating system Rodos.

ID: REQ-SAFE-02
Text: Safety-critical software shall be designed to prevent undefined behavior.
Justification: Undefined behavior may lead to serious problems.
Fullfilled: -/-. Depending on the user implementation.

ID: REQ-SAFE-03
Text: The software shall use a real-time operating system.
Justification: To meet timing requirements.
Fullfilled: Yes. Corfu relies on the real-time operating system Rodos.

B.3 Requirements of Applying Model-Driven Development

ID: REQ-MDD-01
Text: The software framework shall provide a way to define the structure of the software formally.
Justification: The model is the source of truth in the development process.
Fullfilled: Yes. Corfu comes with a model that allows engineers to define the structure of onboard software.

ID: REQ-MDD-02
Text: A generator shall create source code with the information from the model(s)

Justification: This puts the "-driven" into MDD.
Fullfilled: Yes. Corfu comes with a code generator.

ID: REQ-MDD-03
Text: The model definition shall have a concise and easy format for users.
Justification: Users should quickly find their way around the format.
Fullfilled: Yes. Corfu uses YAML.

ID: REQ-MDD-04
Text: The model definition shall use a format that can easily processed with automatic tools.
Justification: Users might implement their own tools for specific model verification.
Fullfilled: Yes. Corfu uses YAML, which is supported by many libraries for most programming languages.

ID: REQ-MDD-05
Text: The model information shall be presented graphically.
Justification: Graphical model representations help developers find their way around the software structure.
Fullfilled: Yes. Corfu generates HTML files and diagrams for interactively navigating through the software configuration.

B.4 Requirements of Model Feedback Features

ID: REQ-MF-01
Text: The feedback process shall create machine-readable files that contain information from software artifacts
Justification: The information must be processable by automatic tools.
Fullfilled: Yes. Corfu stores the information into an SQLite database for easy access for other tools.

ID: REQ-MF-02
Text: The process shall extract the call graph from the source code.
Justification: For example, this is useful for determining maximum stack sizes.
Fullfilled: Yes. Corfu's code analyzer does this.

ID: REQ-MF-03
Text: The process shall extract the maximum stack sizes of functions.
Justification: Useful for the determination of maximum stack sizes.
Fullfilled: Yes. Corfu's code analyzer does this.

ID: REQ-MF-04
Text: The process shall extract the parameters and location of event messages.
Justification: Necessary for moving logging messages out of the code (see 8.4.5).
Fullfilled: Yes. Corfu's code analyzer does this.

ID: REQ-MF-05
Text: The process shall be designed to work iteratively.
Justification: After modifying the model, the framework has to re-generate code .
Fullfilled: Yes. Corfu's process does this.

ID: REQ-MF-06
Text: Creating generated code shall be non-destructive towards user-written code.
Justification: This is necessary for an iterative process.
Fullfilled: Yes. The generated code is stored separately from the user implementation.

B.5 Requirements of Embedded Software

ID: REQ-EMB-01
Text: Embedded software shall consume only little resources in memory and computation power.
Justification: Embedded software often runs on microcontrollers with minimal resources.
Fullfilled: Yes. Corfu does not introduce much resource footprint.

C

Detailed Comparison of Static Memory Usage

This chapter contains some diagrams about memory usage of different software aspects. They compare the memory usage for our classical implementation and Corfu's approach.

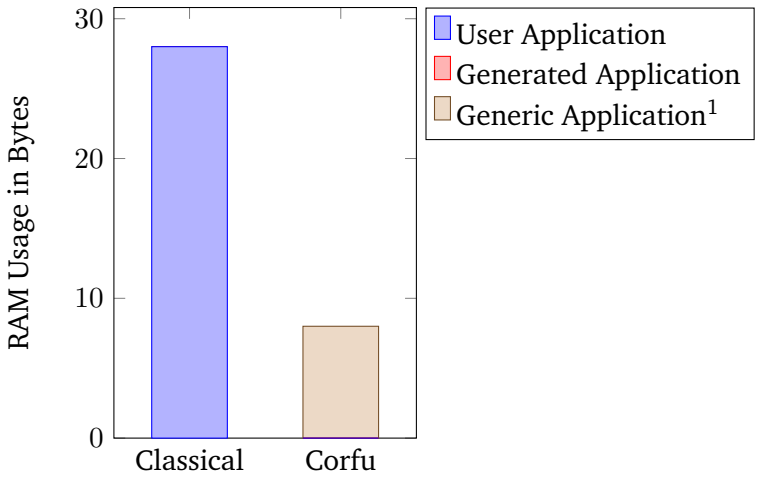


Figure C.1.: Comparison of static memory usage of applications between a classical implementation and Corfu

[1] without telecommand handling

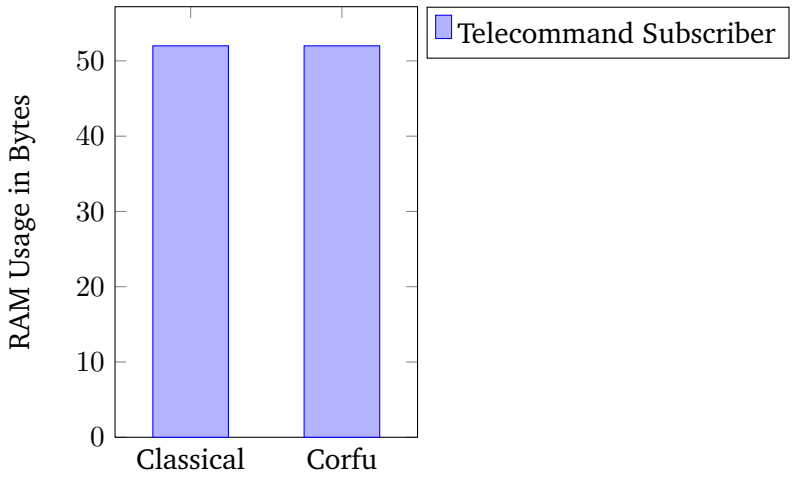


Figure C.2.: Comparison of static memory usage of synchronous telecommand handling between a classical implementation and Corfu

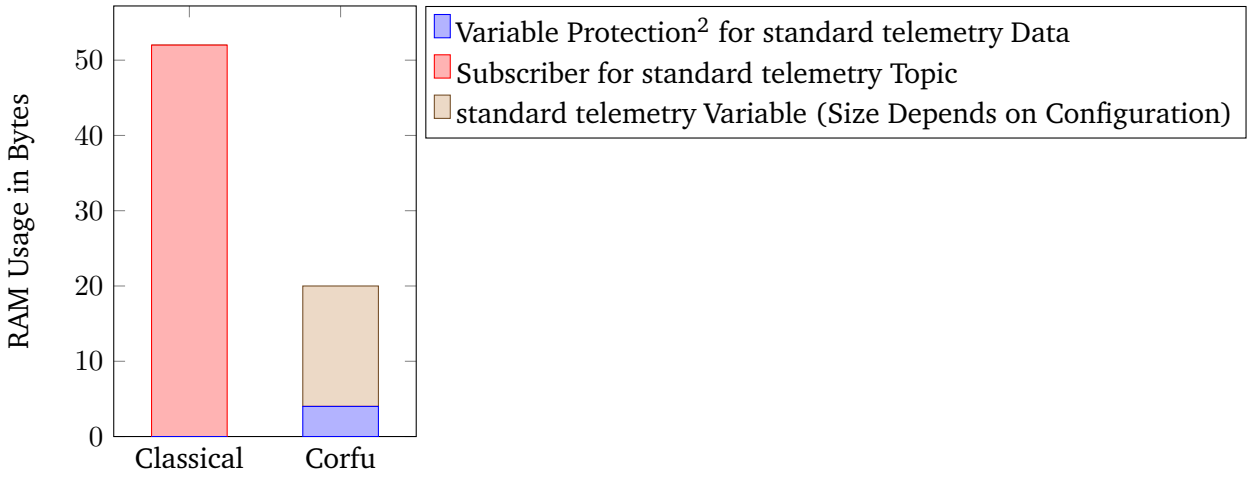


Figure C.3.: Comparison of static memory usage of synchronous telecommand handling between a classical implementation and Corfu

[2]for thread-safety

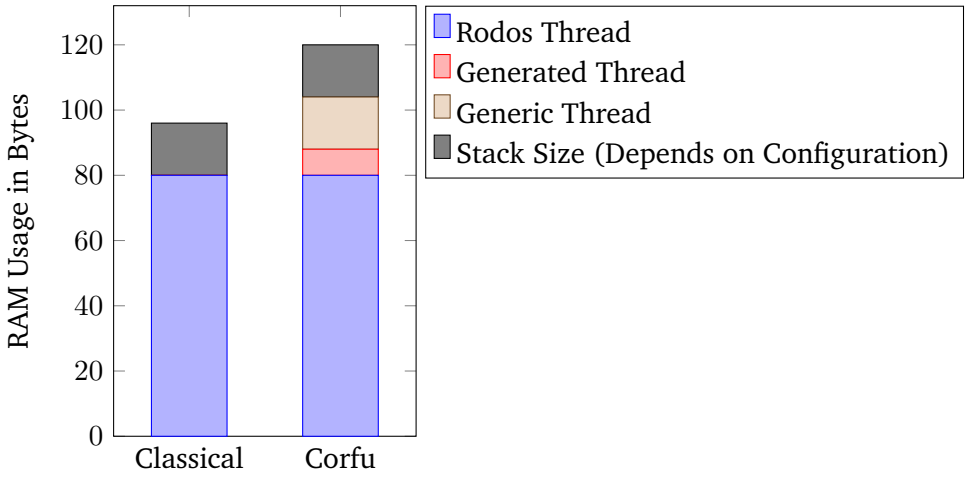


Figure C.4.: Comparison of static memory usage of periodic threads between a classical implementation and Corfu

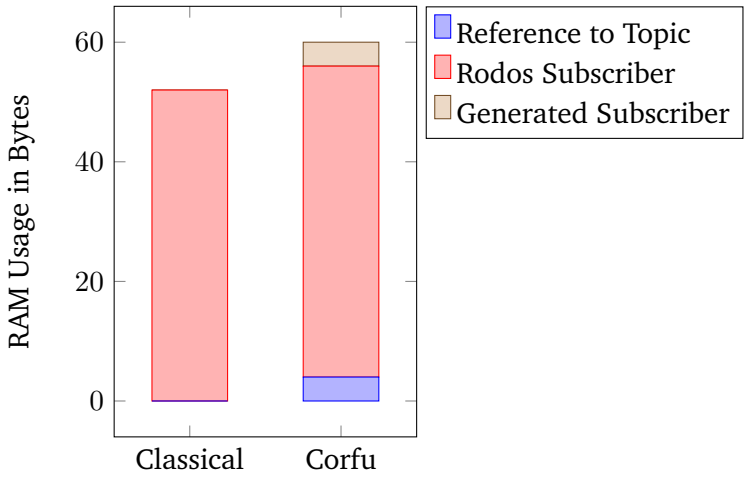


Figure C.5.: Comparison of static memory usage of topic subscription between a classical implementation and Corfu

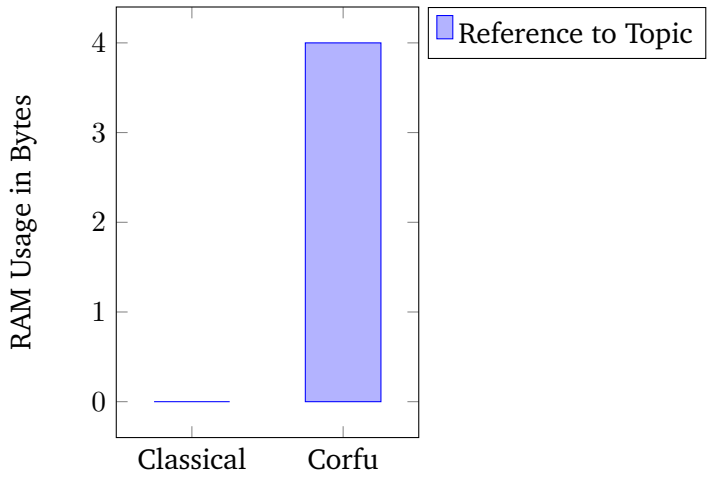


Figure C.6.: Comparison of static memory usage of topic publication between a classical implementation and Corfu

Bibliography

[1] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, 2004

[2] D. Ashley: *Foundation Dynamic Web Pages with Python*, Apress, 2020.

[3] Automotive Open System Architecture: *Guidelines for the use of the C++14 language in critical and safety-related systems*, 2017.

[4] J. Backus, R. Beeper, S. Best, R. Goldberg, L. Haibt, H. Herrick, R. Nelson, D. Sayre, P. Sheridan, H. Stern, I. Ziller, R. Hughes, and R. Nutt: *The FORTRAN Automatic Coding System*, International Workshop on Managing Requirements Knowledge, 1957.

[5] K. Balasubramanian, A. Krishna, E. Turkay, J. Parsons, A. Gokhale, and D. Schmidt *Applying model-driven development to distributed real-time and embedded avionics systems*, International Journal of Embedded Systems, Vol. 2, 2006.

[6] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hung, R. Jeffries, J. Kern, B. Marick, R. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas: *Manifesto for Agile Software Development*, https://agilemanifesto.org/, 2001.

[7] I. Bertolotti and G. Manduchi: *Real-Time Embedded Systems: Open-source Operating Systems Perspecitve*, CRC Press, 2012.

[8] J. Bézivin and O. Gerbé: *Towards a precise definition of the OMG/MDA framework*, Annual International Conference on Automated Software Engineering, 2001.

[9] I. Birrer, V. Cechticky, A. Pasetti, and O. Rohlik: *Implementing Adaptability in Embedded Software through Aspect Oriented Programming*, IEEE Mechatronics & Robotics, 2004.

[10] R. Bocchino, T. Canham, G. Watney, L. Reder, and J. Levison: *F Prime: An Open-Source Framework for Small-Scale Flight Software Systems*, Small Satellite Conference, 2018.

[11] G. Box and N. Draper: *Empirical Model-Building and Response Surfaces*, Wiley, 1987.

[12] M. Brambilla, J. Cabot, and M. Wimmer: *Model-Driven Software Engineering in Practice*, Second Edition, Morgan & Claypool, 2017.

[13] J. vom Brocke, A. Hevner, and A. Maedche: *Design Science Research. Cases*, Springer, 2020.

[14] S. Brüggemann, C. Prause: *Status Quo of Agile Software Development in the European Institutional Space Flight*, Deutscher Luft- und Raumfahrtkongress, 2018.

[15] T. Büchner: *Introspektive modellgetriebene Softwareentwicklung*, Ph.D. Thesis, 2007.

[16] California Polytechnic State University: *CubeSat Design Specification* Revision 13, 2014.

[17] Consultative Committee for Space Data Systems: *Overview of Space Link Protocols*, CCSDS 130.0-G-1, 2001.

[18] Consultative Committee for Space Data Systems: *Space Packet Protocol*, CCSDS 133.0-B-2, 2020.

[19] V. Cechticky, P. Chevalley, A. Pasetti, and W. Schaufelberger: *A Generative Approach to Framework Instantiation*, Generative Programming and Component Engineering, 2003

[20] V. Cechticky, A. Pasetti, and W. Schaufelberger: *The Adaptability Challenge for Embedded Control System Software*, IFAC Proceedings Volume 38, Issue 1, 2005.

[21] V. Cechticky, R. Ottensamer, and A. Pasetti: *Flight software development for the cheops instrument with the cordet framework*, Data Systems in Aerospace, 2015.

[22] C. Coelho, O. Koudelka, and M. Merri: *NanoSat MO Framework: Achieving On-board Software Portability*, SpaceOps Conference, 2016.

[23] C. Coelho, O. Koudelka, and M. Merri: *NanoSat MO framework: When OBSW turns into apps*, IEEE Aerospace Conference, 2017.

[24] D. Comer: *Operating System Design*, CRC Press, 2011.

[25] F. Dannemann and F. Greif: *Software Platform of the DLR Compact Satellite Series*, 4S Symposium, 2014.

[26] F. Dannemann: *Unified Monitoring for Spacecrafts* Dissertation, 2015.

[27] F. Davoli, C. Kourogiorgas, M. Marchese, A. Panagopoulus, and F. Patrone: *Small satellites and CubeSats: Survey of structures, architectures, and protocols*, International Journal of Satellite Communications and Networking, 2018.

[28] M. Dowson: *The ARIANE 5 Software Failure*, ACM SIGSOFT Software Engineering Notes, 1997.

[29] European Cooperation for Space Standardization: *Space Project Management*, ECSS-M-ST-10C, 2009.

[30] European Cooperation for Space Standardization: *Telemetry and telecommand packet utilization*, ECSS-E-ST-70-41C, 2016.

[31] European Cooperation for Space Standardization: *Agile software development handbook*, ECSS-E-HB-40-01A, 2020.

[32] J. Eickhoff: *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*, Springer, 2012.

[33] D. Evans and M. Merri: *OPS-SAT: A ESA nanosatellite for accelerating innovation in satellite control*, SpaceOps, 2014.

[34] M. Faisal and S. Montenegro: *Porting a Real-Time Objected Oriented Dependable Operating System (RODOS) on a customizable system-on-chip*, International Scientific and Technical Conference on Computer Sciences and Information Technologies, 2017.

[35] J. Favre: *Foundations of Model (Driven) (Reverse) Engineering: Models*, Language Engineering for Model-Driven Software Development, 2004.

[36] J. Fernandez and C. Hernandez: *Practical Model-Based Systems Engineering*, Artech House, 2019.

[37] I. Goldstein, T. Kopelowitz, M. Lwenstein, and E. Porat: *Conditional Lower Bounds for Space/Time Tradeoffs* Workshop on Algorithms and Data Structures, 2017.

[38] B. Grzesik, T. Baumann, T. Walter, F. Flederer, F. Sittner, E. Dilger, S. Gläsner, J. Kirchler, M. Tedsen, S. Montenegro, and E. Stoll: *InnoCube A Wireless Satellite Platform to Demonstrate Innovative Technologies*, Aerospace, 2021.

[39] M. Hamilton and the MIT Instrumentation Laboratory: *Source Code of the Apollo Guidance Computer*, 1969.

[40] H. Heidt, J. Puig-Suari, A. Moore, S. Nakasuka, and R. Twiggs: *CubeSat: A new Generation of Picosatellite for Education and Industry Low-Cost Space Experimentation*, Small Satellite Conference, 2000.

[41] G. Holzmann: *The SPIN Model Checker*, Addison-Wesley, 2003.

[42] G. Holzmann: *The Power of 10: Rules for Developing Safety-Critical Code*, Computer Volume 39 Issue 6, 2006

[43] IEEE Standards Board: *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, IEEE Std 982.2-1988, 1988

[44] IEEE and the Open Group: *POSIX.1-2017*, The Open Group Base Specification Issue 7, 2018.

[45] ISO/IEC JTC 1/SC 22: *ISO/IEC 14882:2003 Programming Languages — C++*, ISO, 2003.

[46] ISO/IEC JTC 1/SC 22: *ISO/IEC 14882:2014 Programming Languages — C++*, ISO, 2014.

[47] ISO/IEC JTC 1/SC 22: *ISO/IEC 14882:2020 Programming Languages — C++*, ISO, 2020.

[48] JPL Special Review Board: *Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions*, 2000.

[49] A. Kleppe, J. Warner, and W. Bast: *MDA explained*, Addison-Wesley, 2003.

[50] V. Khorikov: *Unit Testing*, Manning, 2020.

[51] D. Knuth: *The Art of Computer Programming*, Third Edition, Addison-Wesley, 1997.

[52] J. Langr: *Modern C++ Programming with Test-Driven Development*, Pragmatic Bookshelf, 2013.

[53] G. Le Lann: *An Analysis of the Ariane 5 Flight 501 FAilure — A System Engineering Perspective*, International Conference and Workshop on Engineering of Computer-Based Systems, 1997.

[54] M. Li, Z. Shang, Q. Hu, G. Yang, Y. Li, and F. Sun: *Anaylysis and Testing of Key Performance Indexes of Vxworks in Real-Time System*,

2018.

[55] A. Lill, D. Messmann, M. Langer: *Agile Software Development for Space Applications*, Deutscher Luft- und Raumfahrtkongress, 2017.

[56] A. Lill, T. Zwickl, C. Costescu, L. Patzwahl, C. Soare, and M. Langer: *Agile Mission Operations in the CubeSat Project MOVE-II*, SpaceOps Conference, 2018.

[57] The LLVM Team: *Clang: a C language family frontend for LLVM*, https://-clang.llvm.org/, retrieved September 6th, 2021.

[58] The LLVM Team: *Clang 13 documentation: LibTooling*, https://-clang.llvm.org/docs/LibTooling.html, retrieved September 6th, 2021.

[59] Lockheed Martin Corporation: *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*, 2005.

[60] J. Lyons: *Ariane 5: Flight 501 Failure Report By The Inquiry Board*, 1996.

[61] D. Mathur, B. Edwards, J. Goldstein, H. Nguyen, J. Pine, B. Plante, and J. Thacker: *An Approach for Designing Reusable, Embedded Software Components for Spacecraft Flight Instruments*, IEEE Real Time and Embedded Technology and Applications Symposium, 2005.

[62] G. Di Mauro, M. Lawn, and R. Bevilacqua: *Survey on Guidance Navigation and Control Requirements for Spacecraft Formation-Flying Missions*, Journal of Guidance, Control, and Dynamics, 2018.

[63] D. McComas, S. Strege, J. Wilmot: *Core Flight System (cFS) Community: Providing Low Cost Solutions for Small Spacecraft*, Engineering, 2015.

[64] S. Mellor, K. Scott, A. Uhl, and D. Weise: *MDA Distilled*, Addison-Wesley, 2004.

[65] The Motor Industry Software Reliability Association: *MISRA C:1998 Guidelines for the use of the C language in critical systems*, 1998.

[66] The Motor Industry Software Reliability Association: *MISRA C:2004 Guidelines for the use of the C language in critical systems*, 2004.

[67] The Motor Industry Software Reliability Association: *MISRA C++:2008 Guidelines for the use of the C++ language in critical systems*, 2008.

[68] S. Montenegro and F. Dannemann: *RODOS - Real Time Kernel Design for Dependability*, Data Systems in Aerospace, 2009.

[69] S. Montenegro, V. Petrovic, and G. Schoof: *Network Centric Systems for Space Applications*, International Conference on Advances in Satellite and Space Communications, 2010.

[70] C. Nichols: *The Rust Programming Language*, No Starch Press, 2019.

[71] A. Nyßen: *Model-Based Construction of Embedded & Real-Time Software — A Methodology of Small Devices*, Ph.D. Thesis, 2009.

[72] M. Panunzio and T. Vardanega: *A Component Model for On-board Software Applications*, Conference on Software Engineering and Advanced Applications (EUROMICRO), 2010

[73] A. Pasetti, W. Pree, J. Terraillon, and T. van Oberbeek: *An Object-Oriented Component-Based Framework for On-Board Software*, Data Systems in Aerospace, 2001.

[74] A. Pasetti: *Software Frameworks and Embedded Control Systems*, Springer, 2002.

[75] A. Pasetti and W. Pree: *A Component Framework for Satellite On-Board Software*, Digital Avionics Systems Conference, 1999.

[76] R. Plauché: *Building Modern Cross-Platform Flight Software for Small Satellites*, Small Satellites Conference, 2017.

[77] M. Prochazka, R. Ward, P. Tuma, P. Hnetynka, and J. Adamek: *A Component-Oriented Framework for Spacecraft On-Board Software* Data Systems in Aerospace, 2008.

[78] J. Qin, N. Yang, Y. Wang, J. Yang, and J. Du: *A Model-Driven Development Framework for Satellite On-Board Software*, International Conference on Wireless and Satellite Systems, 2020.

[79] S. Rasheed, J. Dietrich, and A. Tahir: *Laughter in the Wild: A Study into DoS Vulnerabilities in YAML Libraries*, IEEE International Conference On Trust, Security And Privacy In Computing And Communications, 2019.

[80] A. Rodríguez, F. Ferrero, E. Alaña, A. Jung, M. Panunzio, T. Vardanega, and A. Grenham: *The Component Layer of COrDET On-Board Software Architecture* Data Systems in Aerospace, 2012.

[81] J. Rothenberg: *The Nature of Modeling*, Artificial Intelligence, Simulation and Modeling, 1989.

[82] D. Schmidt: *Model-Driven Engineering*, IEEE Computer, 2006.

[83] P. Scholz: *Softwareentwicklung eingebetteter Systeme*, Springer, 2005.

[84] G. Sebestyen, S. Fujikawa, N. Galassi, and A. Chuchra: *Low Earth Orbit Satellite Design*, Springer, 2018.

[85] E. Seidewitz: *What Models Mean*, IEEE Software, 2003.

[86] A. Silberschatz, P. Galvin, and G. Gagne: *Operating System Concepts Essentials*, Wiley, 2010.

[87] F. Sittner, C. Liman, G. Schulze, H. Schülein, J. Schmieder, J. Tischhöfer, M. Busch, and S. Montenegro: *Creating a Setup to Assess the Use of Virtual Reality for Mission Control*, Small Satellite Conference, 2021.

[88] T. Stahl, M. Völter, J. Bettin, A. Haase, S. Helsen, K. Czarnecki, and B. von Stockfleth: *Model-Driven Software Development*, Wiley, 2006.

[89] T. Stahl, M. Völter, S. Efftinge, and A. Haase: *Modellgetriebene Softwareentwicklung : Techniken, Engineering, Management*, dpunkt.verlag, 2., aktualisierte und erweiterte Auflage, 2007.

[90] B. Stroustrup: *The C++ Programming Language*, Addison-Wesley, 4th Edition, 2013

[91] B. Stroustrup: *A Tour of C++*, Addison-Wesley, 2018.

[92] M. Sweeting: *UoSAT microsatellite missions*, Electronics & Communication Engineering Journal, 1992.

[93] M. Sweeting: *Modern Small Satellites — Changing the Economics of Space*, Proceedings of the IEEE Vol. 106, 2018.

[94] P. Szécsi, G. Horváth, and Z. Porkoláb: *Improved Loop Execution Modeling in the Clang Static Analyzer*, Acta Cybernetica, 2020.

[95] A. Tanenbaum: *Modern Operating Systems*, Prentice Hall, 2007.

[96] J. Teich and C. Haubelt: *Digitale Hardware/Software-Systeme*, Springer, 2007.

[97] J. Terraillon, A. Jung, P. Arberet, S. Montenegro, A. Rossignol, G. Garcia, J. Li, A. Rodriguez, S. Mazzini, P. Hougaard, S. Fowell, M. Ferraguto, and M. Panunzio: *Space On-board Software Reference Architecture*, Data Systems In Aerospace, 2010.

[98] J. Trimble: *Agile: From Software to Mission System*, SpaceOps Conference, 2016.

[99] K. Umann and Z. Porkoláb: *Detecting Uninitialized Variables in C++*, Acta Cybernetica, 2020.

[100] C. Walls: *Embedded RTOS Design*, Newnes, 2020.

[101] F. Wende: *C++ Data Layout Abstractions through Proxy Types*, International Conference on Engineering of Complex Computer Systems, 2019.

[102] K. Wiegers and J. Beatty: *Software Requirements*, Third Edition, Microsoft Press, 2013.

[103] R. Wieringa: *Design Science Methodology*, Springer, 2014.

[104] J. Wilmot: *A core flight software system*, International Conference on Hardware/Software Codesign and System Synthesis, 2005.

[105] J. Wilmot: *A core plug and play architecture for reusable flight software systems*, IEEE International Conference on Space Mission Challenges for Information Technology, 2006.

[106] Wind River Systems Inc: *VxWorks 7 Datasheet*, 2019.

[107] F. Xiaocong: *Real-Time Embedded Systems*, Newnes, 2015.

[108] C. Ziemke, T. Kuwahara, and I. Kossev: *An integrated development framework for rapid development of platform-independent and reusable satellite on-board software*, Acta Astronautica 69, 2011.

Acknowledgement

A big thank you goes out to everyone who supported me to realize this work. First, I want to thank my supervisor Sergio Montenegro. With his experience, he showed me the right track for developing Corfu. His hints and comments were always beneficial. He managed to create a great chair with a fantastic working atmosphere, which helped me work efficiently.

Thank you to my colleagues at the InnoCube project, Erik Dilger, Felix Sittner, Thomas Walter, and Tom Baumann, for their trust in my framework. It will be Corfu's first orbit experience. At this point, special thanks to Felix and Tom for testing and driving for new features of Corfu. Without Ludwig Ostermayer, I probably would have never started my academic career.

My dearest Kerstin, thank you for keeping my back free and having so much patience; I love you. Also, thank you, Marlene; your smile always helped me to recreate. Recreational were also all the Sunday afternoons at my parents; thank you for always the warm welcome. You have made it possible to grow to my full potential; this is also your result.

Research in Aerospace Information Technology

This monograph series is published by the Chair of Aerospace Information Technology (Informatik VIII) of the University of Würzburg and presents innovative research regarding avionic systems for aerospace and terrestrial applications as well as the technology transfer between both fields.
The main research focus is on the development of reliable soft- and hardware for embedded applications that allow the autonomous operation of unmanned systems in challenging environments. This includes the development of new technologies such as wireless communication methods, distributed sensing and control strategies, sensor fusion algorithms, novel navigation methods and concepts for dependable software targeting the irreducible complexity.
Another research focus is on cooperative tasks of multi-agent systems, including homogeneous swarms and arbitrary heterogeneous constellations.
The developed technologies are deployed in numerous real-world applications such as small satellite systems, distributed sensor networks, unmanned aerial vehicles for extreme environments and other experimental platforms.

Herausgeber:

Prof. Dr. Sergio Montenegro

ISSN: 2747-4828

Zitiervorschlag:
Flederer, Frank (2021): CORFU – An Extended Model-Driven Framework for Small Satellite Software with Code Feedback. Research in Aerospace Technology, 2. DOI: 10.25972/OPUS-24981