TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Software Science

Dan Rodionov 153074IAPM

IMPLEMENTING TTÜ NANOSATELLITE COMMUNICATION PROTOCOL USING TASTE TOOLSET

Master's thesis

Supervisor: Evelin Halling
MSc

Tallinn 2017

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Dan Rodionov 153074IAPM

TTÜ NANOSATELLIIDI KOMMUNIKATSIOONIPROTOKOLLI IMPLEMENTEERIMINE KASUTADES TASTE TÖÖRIISTAKOMPLEKTI

magistritöö

Juhendaja: Evelin Halling
MSc

Tallinn 2017

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Dan Rodionov

15.05.2017

Abstract

The purpose of this thesis is to use the TASTE Toolset for modeling the communication protocol of TTÜ-Mektory Student Satellite and generate C code based on that model. Tools in the TASTE Toolset can be used for validating the C code and verifying the requirements of TTÜ-Mektory Student Satellite communication protocol.

The purpose is achieved by modelling a part of the TTÜ-Mektory Student Satellite communication protocol. In the scope of this thesis, the communication between ground station and satellite on-board computer is modeled. The system is modeled in parts, which are designed so that they could be used both on the ground and on the satellite. To validate the generated code, the modeled systems are connected to each other and used to represent the communication process.

Before modeling the system, the description of TTÜ-Mektory Student Satellite communication protocol is studied. Based on that description, class diagrams for communication structure and sequence diagrams for the communication process are created. After the communication protocol has been described as diagrams, the system is modeled in the TASTE Toolset. The data types are described in ASN.1 notation and the components of the system are modeled in SDL. When the system has been described, Ada code for the system components is generated. The system architecture is described and based on that description, C code is generated.

The communication process is tested by using the tools in the TASTE Toolset. A GUI component is used to represent the mission control system, which can be used for interaction with the system. Based on the sequence diagrams of different communication scenarios, raw data is sent to the system and the data received from the system is compared to the requirements in the TTÜ-Mektory Student Satellite communication protocol.

As a result of this thesis, a representation of the communication protocol is created and the requirements of the TTÜ-Mektory Student Satellite communication protocol are

verified. The created system can be used in the communication process of the TTÜ-Mektory Student Satellite both on ground and on the satellite. The result of this thesis will be used as a case study in ESA project that integrates TASTE Toolset with QGen.

This thesis is written in English and is 37 pages long, including 9 chapters, 14 figures and 8 tables.

Annotatsioon

TTÜ nanosatelliidi kommunikatsiooniprotokolli implementeerimine kasutades TASTE tööriistakomplekti

Lõputöö eesmärgiks on kasutada TASTE tööriistakomplekti TTÜ ja Mektory koostöös valmiva tudengisatelliidi kommunikatsiooniprotokolli modelleerimiseks ja loodud mudeli põhjal C koodi genereerimiseks. Selleks, et veenduda genereeritud C koodi ja kommunikatsiooniprotokolli nõuete korrektsuses, kasutatakse TASTE tööriistakomplekti poolt pakutavaid tööriistu.

Eesmärgini jõutakse modelleerides osa tudengisatelliidi kommunikatsiooniprotokollist. Antud töö lõikes vaadeldakse suhtlust maajaama ja satelliidi pardakompuutri vahel. Süsteem modelleeritakse osadena mis on disainitud nii, et neid saaks kasutada nii maa kui ka satelliidi peal. Selleks, et genereeritud koodi valideerida, ühendatakse modelleeritud süsteemid teineteisega ja luuakse kahe süsteemi vaheline suhtlus.

Enne suhtluse modelleerimist uuritakse TTÜ ja Mektory koostöös valmiva tudengisatelliidi kommunikatsiooniprotokolli kirjeldust. Selle kirjelduse põhjal luuakse klassidiagrammid suhtluse struktuuri ja jadadiagrammid suhtlusprotsessi kohta. Pärast kommunikatsiooniprotokolli kirjeldamist diagrammide abil modelleeritakse süsteem kasutades TASTE tööriistakomplekti. Andmetüübid kirjeldatakse kasutades ASN.1 notatsiooni ja süsteemikomponendid modelleeritakse keeles SDL. Kui süsteem on kirjeldatud, genereeritakse süsteemikomponentide kood keeles Ada. Seejärel kirjeldatakse süsteemi arhitektuur ja genereeritakse kood keeles C.

Kommunikatsiooniprotokolli testitakse kasutades TASTE tööriistakomplekti tööriistu. Kasutajaliidese komponenti kasutatakse missioonijuhtimise kujutamiseks ja selle abil suheldakse süsteemiga. Erinevate suhtlusstsenaariumite ja neid kirjeldavate jadadiagrammide alusel saadetakse andmed süsteemile ning süsteemilt tagasi saadud andmeid võrreldakse tudengisatelliidi kommunikatsiooniprotokolli nõuetega.

Lõputöö tulemusena luuakse süsteem kommunikatsiooniprotokolli haldamiseks ning verifitseeritakse TTÜ ja Mektory koostöös valmiva tudengisatelliidi kommunikatsiooniprotokolli. Loodud süsteemi saab kasutada tudengisatelliidi projektis nii maa kui ka satellidi poolseks suhtluseks. Lõputöö tulemust kasutatakse katseülesandena ESA projektis, mis integreerib TASTE tööriistakomplekti QGeniga.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 37 leheküljel, 9 peatükki, 14 joonist, 8 tabelit.

List of abbreviations and terms

TASTE	*The ASSERT Set of Tools for Engineering*
ESA	*European Space Agency*
UHF	*Ultra High Frequency*
TTÜ	*Tallinn University of Technology*
SDL	*Specification and Description Language*
UML	*Unified Modeling Language*
ASN.1	*Abstract Syntax Notation One*
AADL	*Architecture Analysis and Design Language*
ICD	*Interface Control Document*
ATV	*Automated Transfer Vehicle*
FP6	*Sixth Framework Programme*
TM/TC	*Telemetry and Telecommand*
CRC	*Cyclic Redundancy Check*
PUS	*Packet Utilization Standard*
AX.25	*Amateur X.25*
UI	*Unnumbered Information*
TBD	*To Be Decided*
P/F	*Poll/Final*
PID	*Protocol Identifier*
BA	*Bus Address*
SRC	*Source*
DST	*Destination*
OBC	*On-Board Computer*
IFCS	*Info Frame Check Sequence*
LSB	*Least Significant Bit*
FCS	*Frame Check Sequence*
TtE	*Time-to-End*
MRT	*Maximum Response Time*

LFN	*Lowest Sequential Frame Number*
TCP/IP	*Transmission Control Protocol/Internet Protocol*

Table of contents

List of figures

List of tables

1 Introduction

TASTE (The ASSERT Set of Tools for Engineering) [1] is a set of software development tools, which can be used to describe software with formal models, verify created models and automatically generate program code from these models [2]. TASTE is suitable for describing real-time heterogeneous systems.

The subject of the thesis is software modelling and validating with TASTE tools. The reason for choosing TASTE tools is that they have been developed in collaboration with ESA (European Space Agency) and they support the whole life-cycle of space system software development [3].

As a software system, part of the TTÜ-Mektory Student Satellite Space System is going to be used. The TTÜ-Mektory Student Satellite Space System is shown in Figure 1.

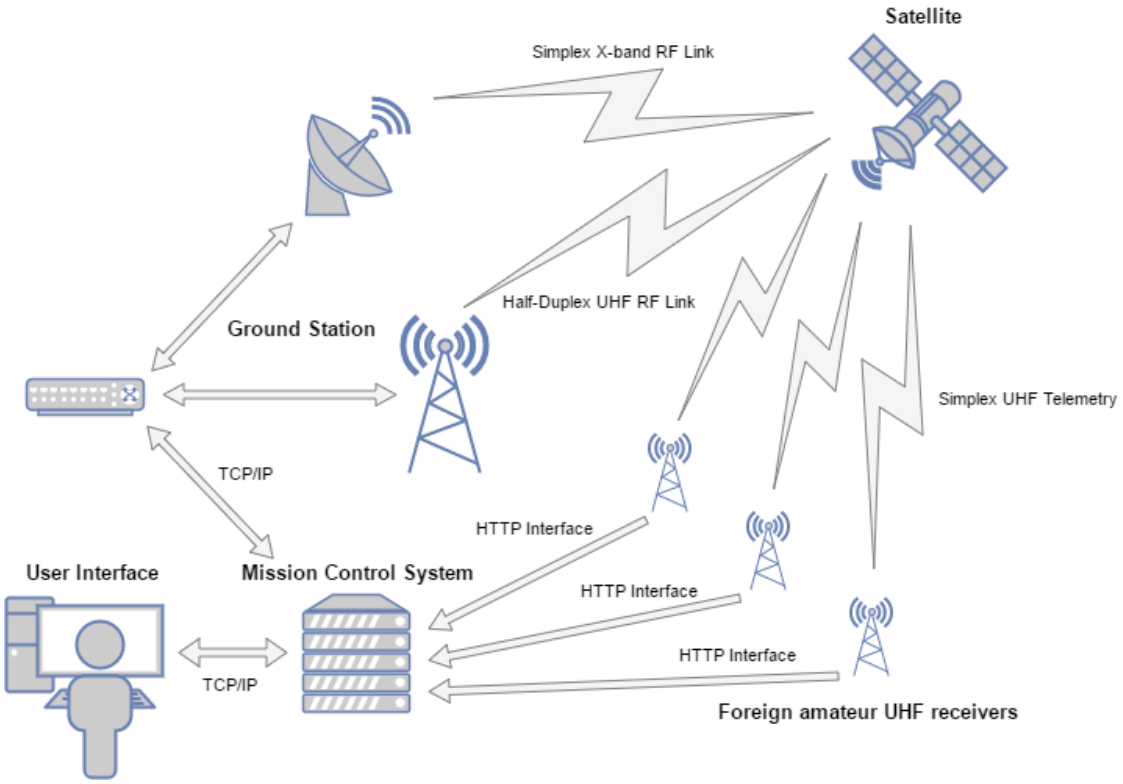


Figure 1. TTÜ-Mektory Student Satellite Space System.

TTÜ-Mektory Student Satellite Space System consists of ground and space segment. Ground segment can be divided into following elements:

- ground station,
- mission control system,
- foreign amateur UHF (Ultra High Frequency) receivers.

Space segment consists of a satellite.

In this thesis, the communication protocol of TTÜ-Mektory Student Satellite is modeled. The communication protocol is used in communication between the ground station and satellite as shown in Figure 2.

Figure 2. TTÜ-Mektory Student Satellite communication protocol.

When modeling the software system, the thesis is looking for answers to the questions regarding the description of communication protocol and usability of the TASTE Toolset. The thesis is checking whether the description of communication protocol requirements is sufficient. In case it is insufficient for modeling, the ways for improving the description of communication protocol are looked into. As the modeling results with generated code, the usability of the code in the TTÜ-Mektory satellite program is evaluated. Based on that evaluation a decision will be made regarding the sensibility of using the same method for creating other software components of the system.

In order to verify the correctness of communication protocol, system architecture and communication channels must be defined. To describe components as processes and state machines, SDL (Specification and Description Language) formal language and modelling options of TASTE tools are going to be used. After describing the components, the code that has been generated based on the model can be validated with tests.

2 TASTE Toolset

2.1 Background

TASTE is a set of software development tools, which is similar to the UML (Unified Modeling Language), apart from the code generation functionality. UML is a fully functional graphical editor, but it lacks the ability to support the development process of a system [4].

The TASTE Toolset is based on two modeling languages: ASN.1 (Abstract Syntax Notation One) and AADL (Architecture Analysis and Design Language). It was developed with the purpose to build optimal systems, which use manually or automatically produced heterogeneous components and run based on a pre-defined specification [5]. It was created in 2008 with the support from ESA.

The TASTE Toolset is often used for developing systems, which have some of the following characteristics:

- limited resources,
- time constraints,
- varying nature (laws, resources, fault detection),
- shared development,
- hardware communication,
- heterogeneous hardware,
- physical distribution,
- autonomous operation,
- physical inaccessibility [2].

With TASTE it is possible to connect all of the system components and deploy them on a specific target. The developed code is transparent and there is no need for message formatting and decoding, system configuring or resource management debugging, as those tasks are already automated [2]. QGen code generator can be used to generate C code from the TASTE model [6]. That code can be verified according to the specification in order to assure system validity.

As TASTE Toolset is used for developing software components of heterogeneous nature, it enables software development in different languages. The supported languages include Python, Simulink, MSC, Ada, SMP2, C, VHDL, SCADE and SDL [2].

2.1.1 OpenGEODE

SDL (Specification and Description Language) is a modelling language for describing systems in the form of state machines [7]. It has been established by ITU-T under reference Z100 and is mostly used in the telecommunication industry. SDL has established semantics and is useful for describing embedded, real-time systems.

TASTE Toolset includes a graphical SDL editor OpenGEODE that can be used for editing processes and procedures. It features description of hierarchical and parallel states and its model can be used for generating Ada code. As OpenGEODE supports pre-defined ASN.1 data types, then it can be efficiently used for model checking [8].

2.1.2 ASN.1 and AADL

In a heterogeneous environment, the system communication is based on ASN.1 technology. It is an ISO/IEC and ITU-T standard, that defines the notation for describing data structures [9]. The messages that are defined using ASN.1, can be either abstract or physical. As the tools translate the messages from one form to another, there is no risk that the data will be interpreted incorrectly.

The ASN.1 technology is favourable when working with embedded systems, as the complexity is small and the learning curve is not steep. It is relying on an ISO based standard and thus has been used for years in areas such as banking transactions, aeronautical communication networks etc [2]. In order to support safety-critical systems, an open-source ASN.1 compiler ASN1SCC has been developed, which supports static memory, automatic statement coverage, SPARK/Ada annotations, integration with

legacy systems and automatic ICDs (Interface Control Document) [9, 10]. ASN1SCC is used for parsing ASN.1 grammar and converting it to C or Ada declarations and functions which can be used for encoding/decoding these types to/from binary streams [11].

In addition to ASN.1, another technology being used is AADL. When defining the architecture of a TASTE system, AADL can be used to represent it in a textual format. The definition consists of functional blocks that have some non-functional attributes. As with ASN.1 technology, the AADL is simple to understand.

ASN.1 and AADL together define a model that describes the system completely. This description includes types manipulated, interfaces of processes and threads, connection topology and flow of information and interaction [4].

2.2 Process

The ASSERT process for software development with the TASTE Toolset consists of the following steps:

1. System modelling phase, where software is abstracted;
2. Transformation phase, which results with a real-time software;
3. Feasibility analysis phase, where properties are statically verified;
4. Code generation phase, where binary files are generated [4].

System modelling is conducted in the *interface view* editor and the specification is defined in the *deployment view* editor. Both are graphical tools, which are used for describing the logical interactions and hardware architecture of the system. To support large scale architecture, both of these editors have the functionality of grouping functions into hierarchical containers.

When the logical interactions and hardware architecture have been described, the result is submitted to a vertical transformation tool. During this automated procedure, software and hardware is generated. As the generation is based on a description, then the result contains all the real-time and distribution properties [4].

The result of the *vertical transformation* is displayed in the *concurrency view* editor, where performance analysis can be conducted. The tools for performing that analysis are Cheddar and dynamic simulator, both of which have been integrated into the TASTE Toolset.

As a last step, an executable application can be generated from the pre-defined functional blocks. This is done using the Ocarina tool, which has multiple choices for code generation based on the system architecture.

2.3 Related work

2.3.1 Current state

The development of the TASTE Toolset was first focused on the development process, rather than choosing which technologies to use [4]. After the development process had been set, modeling languages were selected and integration issues were faced. As a result, a prototype was made which supported all of the system development phases. Soon the product was released as a complete toolset, which was tested among system and software designers [4].

As the solution had many options for system development, it confused the users regarding its functionality [4]. Tool developers had to offer support to the users and help them discover the wide range of functions that the tool had to offer. After facing those issues, the users started seeing the advantages in system development with this particular toolset [4].

2.3.2 Projects

TASTE can be used in different areas from educational purposes to application on operational projects [3]. Although system development for classical Earth-orbiting spacecrafts remains unchanged, then the toolset could be used for developing more complex and challenging systems like formation-flying, deep-space probes, robotics and next generation launchers [3].

The ASN.1 modelling language was evaluated by Astrium in ATV (Automated Transfer Vehicle) program, by retro-engineering a part of it [9]. The AADL technology has been used to build critical real-time systems in the IST-ASSERT project, which is part of the

FP6 (Sixth Framework Programme) of the European Commission [12]. The aim of this project is to provide tools and methods to ease the development of safety-critical systems, such as in space.

There are examples where TASTE Toolset and ASN.1 technology have been used for modelling satellite TM/TC (Telecommand and Telemetry). First example [13] shows the modelling of TM/TC and the second example [14] also includes the use of CRC (Cyclic Redundancy Check) and length fields. Both examples are based on ESA's PUS (Packet Utilization Standard).

TASTE Toolset has been used to develop software for PROBA-3 Coronagraph Instrument, which will be used in future ESA missions for in-orbit demonstration of precise formation flying techniques and technologies [3]. The ASN.1 technology was used to address different integration issues and ensure end-to-end data consistency. The toolset was also used in developing the UPMSat-2 satellite manager software subsystem, where TASTE was used to design the model, facilitate a user interface and implement the manager [15].

It has been also used in a case study to simulate the control of a robotic arm using an exoskeleton [16]. The exoskeleton sent data to the computer, which was running the TASTE generated binary files. Those binary files translated the sensor information and were used to command a 3D model of a robotic arm. In the future, this 3D model could be replaced by a real arm.

2.3.3 Future

Although the TASTE Toolset is ready to be used as is, plans should be made regarding the technical perspectives and the toolset itself [4]. There are many opportunities for supporting the standard languages and offering more openness to the users. Additions could be made to the functionality and the usability of this software in real industrial environments could be improved.

3 Communication protocol

The following paragraph is based on TTÜ-Mektory Nanosatellite TM/TC Protocol Description [17] and AX.25 (Amateur X.25) Amateur Packet-Radio Link-Layer Protocol [18].

The AX.25 protocol is used as the TTÜ-Mektory Student Satellite is communicating over amateur frequencies. Using amateur frequencies has the requirement of message readability, which is why the publicly available AX.25 Amateur Packet-Radio Link-Layer Protocol was the optimal solution.

Communication between the TTÜ-Mektory Student Satellite and ground station is established using the UHF RF interface. An amateur radio station will be used in ground station. It will also have a tracking antenna system for automatic alignment of the ground and satellite antenna.

Parts of the communication protocol, such as bus protocol commands and the affiliated structures that are not in the scope of this thesis, are not described.

3.1 Telemetry and telecommand protocol

In TTÜ-Mektory Student Satellite telemetry and telecommand communication a subset of AX.25 Amateur Packet-Radio Link-Layer Protocol [18] is used.

3.1.1 Frame structure

In terms of frame types, the TTÜ-Mektory Student Satellite supports only AX.25 UI (Unnumbered Information) frames. The repeated frames specified by the AX.25 protocol are not supported. Any other types of frames that are not specified in the communication protocol are ignored by the system.

The communication between the ground station and satellite is in bursts. Each burst consists of a radio header used for synchronization that is followed by a number of AX.25

frames. The structure of a burst is shown in Table 1 and the structure of an AX.25 frame is shown in Table 2.

Table 1. Communication burst structure with parameters.

Radio Header	**AX.25 Frame 1**	**…**	**AX.25 Frame N**
TBD (To Be Decided) bits	200-2208 bits	…	200-2208 bits

Table 2. AX.25 frame structure with parameters.

Flag	**Addr**	**Control**	**PID**	**Info**	**FCS**	**Flag**
8 bits (0x7E)	112 bits	8 bits	8 bits	N (up to 256) x 8 bits	16 bits	8 bits (0x7E)

Communication is initialized by sending a burst of AX.25 frames to the satellite. Satellite responds to the burst with a similar burst that contains the response. The whole transmission is done during an available time window.

When satellite is transferring telemetry, it sends a radio header with a single AX.25 frame. Telemetry transferring is used for monitoring satellite status. As the telemetry frames are publicly available, they can be received by the ground station or radio amateurs.

3.1.1.1 Flag

Flag is used for separating each AX.25 frame. The end flag of a previous frame is the start flag of the next frame. *Flag* is represented by the code 0x7E which means that is consists of six consecutive 1's.

As six consecutive 1's are used as flags, then they can not be present in the frame content. To avoid that, bit stuffing is used before assembling the burst. During bit stuffing, five consecutive 1's are followed by a 0. Before parsing the frames, the 0's are removed for restoration of the data.

When no data is available for transmission, only flags are transmitted.

3.1.1.2 Address field

Address field is used for describing the frame source and destination. As the communication is always between the same ground station and satellite, the frame source

and destination have the same values during each transmission. For sending and receiving data from ground to satellite or satellite to ground, pre-defined *Address* values are used.

When replying to a message received from UHF amateur radio, the address of a sender is used.

Address consists of 56-bit destination station address and 56-bit source station address. The *Address* field structure is shown in Table 3, where the octets containing destination and source address are labelled A1-A14.

Table 3. Address field structure and its contents.

112-bit Address Field													
Destination Address							**Source Address**						
A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14

3.1.1.3 Control field

Control field is used for indicating the type of AX.25 frame. As TTÜ-Mektory Student Satellite is using AX.25 UI frames, then the value of *Control* is fixed to 0x03. P/F (Poll/Final) bit is not used and is set to 0.

3.1.1.4 PID field

PID (Protocol Identifier) field is used for defining the protocol of *Info*. No defined AX.25 protocols are used in communication between ground station and satellite, so the *PID* is fixed to 0xF0.

3.1.1.5 Info field

Info field is used for sending information from ground to satellite or satellite to ground. *Info* consists of two parts:

- inner satellite bus (*BA*, *Code*, *Operation Data* and *IFCS* fields),
- optional authentication (*Auth* field).

The structure of the *Info* field is shown in

Table 4.

Table 4. Info field structure with parameters.

BA	Cntrl	Code	Operation Data	IFCS	Auth
8 bits	8 bits	8 bits	N (up to 250) x 8 bits	16 bits	16 bits

3.1.1.5.1 BA

BA (Bus Address) field is used on the inner satellite bus. It is used for describing sender and receiver. Some *BA* values also define how frames are sent from satellite to ground.

BA consists of 4 bit SRC (Source) and 4 bit DST (Destination) Address. The structure of the *BA* field is shown in Table 5, where the bits containing source and destination address are labelled 7-0.

Table 5. BA field and its contents.

8-bit BA Field							
SRC Address				DST Address			
7	6	5	4	3	2	1	0

3.1.1.5.2 Cntrl

Cntrl field defines the frame type. As also with AX.25 frames, the *Cntrl* is fixed to 0x03 which defines the AX.25 UI frame.

3.1.1.5.3 Code

Code is used for specifying an operation on satellite internal bus. *Code* also defines the *Operation Data* structure and the response type. Each *Operation Data* structure has a specific response.

For example there are different codes for:

- telemetry reading,
- parameter writing,
- various responses.

Operations on satellite internal bus are used in situations where the OBC (On-Board Computer) is inaccessible. As it is a specific scenario, then bus protocol commands, their structures and response types are not described in this thesis.

Besides defining operations on satellite internal bus, some codes are used for communication between ground station and OBC. Codes are sent in a set of frames in one direction and received in another direction.

The structure of these commands is described in 3.1.2.

3.1.1.5.4 Operation data

Operation Data is used for storing data of a specific command. It can be used for satellite bus protocol commands or L3/L4 frames.

The details of satellite bus protocol are not described in this thesis.

When using *Operation Data* field for data of L3/L4 frames, the *BA, Cntrl, Code* and *IFCS* fields of *Info* structure are replaced with *BA, Cntrl, Code* and *FCS* fields of L3/L4 frame. The rest of the data described in a L3/L4 frame is stored in *Operation Data* field.

3.1.1.5.5 IFCS

IFCS (Info Frame Check Sequence) field is used for storing the frame check sequence of satellite internal bus frame. It is calculated over *BA*, *Cntrl*, *Code* and *Operation Data*. Apart from the other fields specified in the communication protocol, the *IFCS* field is transmitted LSB (Least Significant Bit) first. The details of *IFCS* calculation are described in 3.1.1.6.

3.1.1.5.6 Auth

Auth field is optional and used in specific scenarios. When operating with commands that require additional security, the *Auth* is used to verify the source of the data. The *Auth* field is not present on the internal bus as it is removed by the radio module. The frames that have invalid *Auth* values are not sent to the internal bus.

3.1.1.6 FCS

FCS (Frame Check Sequence) field is used for frame validation. The value is calculated according to the ISO 3309 standard recommendations and compared to the value stored in the *FCS* field. It is used to validate the correctness of the frame and avoid corruption.

The polynomial used in the *FCS* calculation is shown in Equation (1).

$$FCS = X^{16} + X^{12} + X^5 + 1 \quad (1)$$

The *FCS* is calculated over *Address*, *Control*, *PID* and *Info* fields. As with *IFCS* field, it is transmitted LSB first.

3.1.2 L3/L4 frames

L3/L4 frames are used for transporting higher layer data between mission control and OBC. L3/L4 frames are put together in mission control, transferred to ground station and transmitted to satellite. Each transmission consists of a burst, which contains a number of AX.25 frames. AX.25 frames carry the information of L3/L4 frames.

When not transmitting data or telemetry, the satellite is always in the listening state. Satellite never initiates a transfer. During ground station to satellite communication, each sent frame is accompanied with the *TtE* (Time-to-End) which indicates the time left to the end of transmission. With each received frame, the timer on the satellite is increased.

When a burst has been received, the AX.25 frames are removed before sending the data to the satellite internal bus. Each frame is acknowledged by the OBC and an acknowledgement frame is sent to the radio. Sent frames are monitored by the radio and the acknowledgement frames are not sent to the ground station. When response is assembled, the source and destination addresses are swapped.

Reception of last frame triggers a response from OBC to ground station. Timer is updated, frames are acknowledged and sent to the ground segment.

The L3/L4 communication diagram is shown in Figure 3.

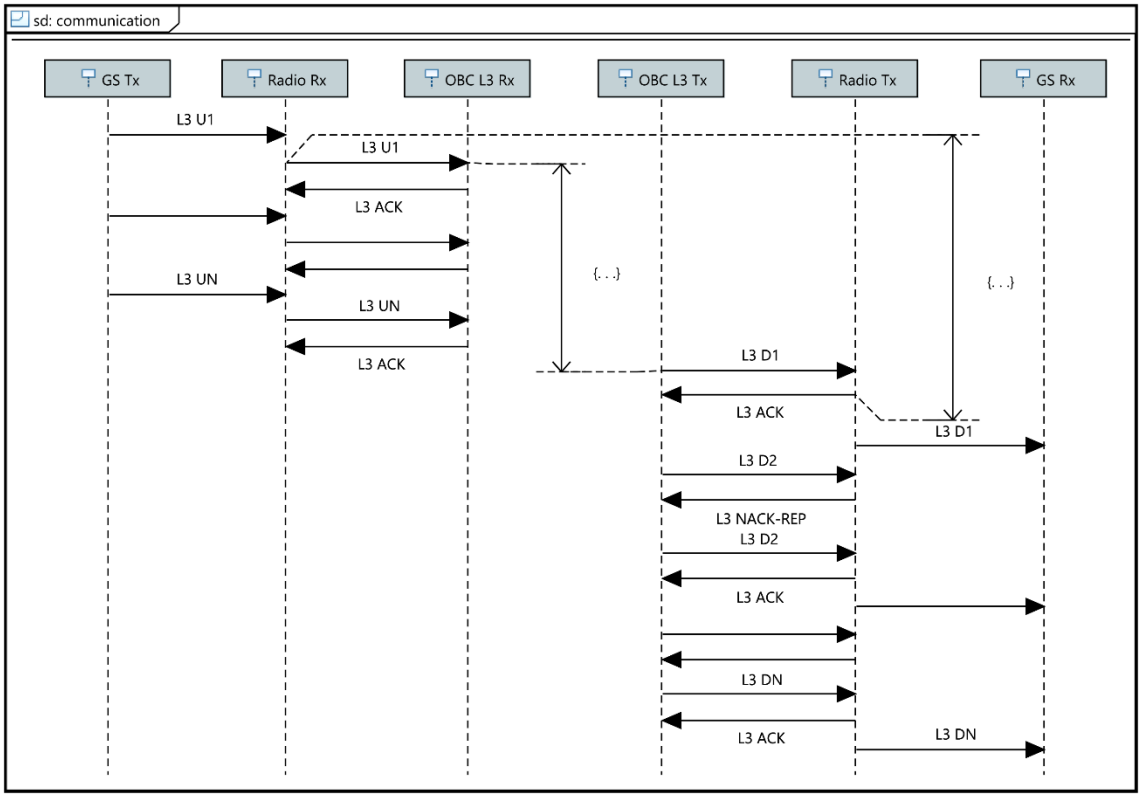


Figure 3. L3/L4 communication diagram.

3.1.2.1 L3 Init/Reset command

The structure of an L3 Init/Reset command is shown in Table 6.

Table 6. Init/Reset command structure with parameters.

BA	Cntrl	Code	TtE	MRT	FCS
8 bits	8 bits (0x03)	8 bits (TBD)	16 bits	16 bits	16 bits

Init/Reset command is sent by the ground station to the satellite in the beginning of each communication session. It could also be sent in the middle of a communication session, when a clean session is needed. During *Init/Reset* command, the frame numbers are set to 0 and the higher layers of protocols are also reset. *Init/Reset* command is also used when communication goes into an abnormal state. For example when frames are not following a logical sequence, the communication is restarted and a clean session is initialised.

3.1.2.2 L3 data frame

The structure of an L3 data frame is shown in Table 7.

Table 7. L3 data frame with parameters.

BA	Cntrl	Code	TtE	MRT	Fnum	Data	FCS
8 bits	8 bits (0x03)	8 bits (TBD)	16 bits	16 bits	8 bits	N (up to 245) x 8 bits	16 bits

L3 data frame is used for carrying data into higher layers of the communication protocol. Each frame includes *TtE*, *MRT* (Maximum Response Time) and *Fnum*. *TtE* indicates the time left to the end of transmission, *MRT* limits the amount of data that satellite is allowed to respond with and *Fnum* is used for frame numbering. Frame numbers are used to reorder the data and request missing frames.

3.1.2.3 L3 data acknowledge

The structure of an L3 data acknowledge frame is shown in Table 8.

Table 8. L3 data acknowledge frame with parameters.

BA	Cntrl	Code	TtE	MRT	LFN	HFN	RRQ_1	…	RRQ_N	FCS
8 bits	8 bits (0x03)	8 bits (TBD)	16 bits	16 bits	8 bits	8 bits	8 bits	8 bits	8 bits	16 bits

L3 data acknowledge frames are used for acknowledgement of L3 data frames. An acknowledge frame is sent once during a burst and it can be located anywhere within the burst.

When L3 frame is received, the L3 stack is checked for missing frames. If the frame number of the received frame is one unit bigger than the previous received frame, the frame number is put into *LFN* (Lowest Sequential Frame Number) field and the frame is forwarded to higher layers. If the frame number is bigger, the buffer is scanned for a frame with missing numbers. If missing frames are found, the *LFN* value is updated and frames are forwarded, if not, the frame is stored.

3.1.2.4 L3 bus acknowledge

L3 bus acknowledgement frames are used on satellite internal bus between radio and OBC. The frames are used to acknowledge the reception of L3 frames and contain the information about the transmission status.

As the satellite internal bus is not a part of this thesis, then L3 bus acknowledgement frames are not used.

4 Modelling

The modeling of the communication protocol began with the study of TTÜ-Mektory Nanosatellite TM/TC Protocol Description [17]. As the main communication is between ground station and OBC, the L3 data with the frame structure was chosen as the project scope. Bus protocol commands were not included in the initial model, as they are only used in a situation where OBC is inaccessible.

The structure of the components that were necessary for communication was described as a class diagram (4.1). After the initial components had been described, the sequence diagrams of different communication scenarios were created. These scenarios included successful communication (4.2.1), partially successful communication (4.2.2) and timeout (4.2.3). Both the class diagrams and sequence diagrams were created with Papyrus [19], which is a UML modeling environment built on Eclipse.

After the class and sequence diagrams had been created, a pre-configured virtual machine image with TASTE [20] was used for system modeling. The virtual machine image was deployed in Oracle VM VirtualBox [21] virtualization environment. The data types were described in ASN.1 and the choice of data types was based on the examples in TASTE V2 Reference Card [22]. Having the data types described, the *interface view* with the initial components was created in TASTE. The logic for each component was modeled in OpenGEODE, which is an SDL editor for TASTE (4.3). Before generating C code, the architecture of TTÜ-Mektory Student Satellite space system was described in the *deployment view*.

4.1 Class diagrams

The data types are described in Figure 4.

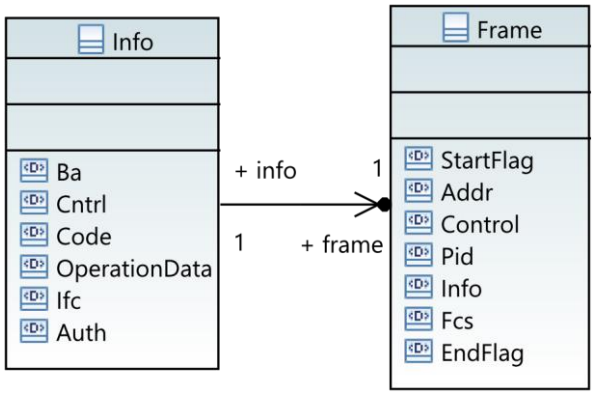


Figure 4. Class diagram of data types.

The components with the data types and corresponding procedure calls are described in Figure 5.

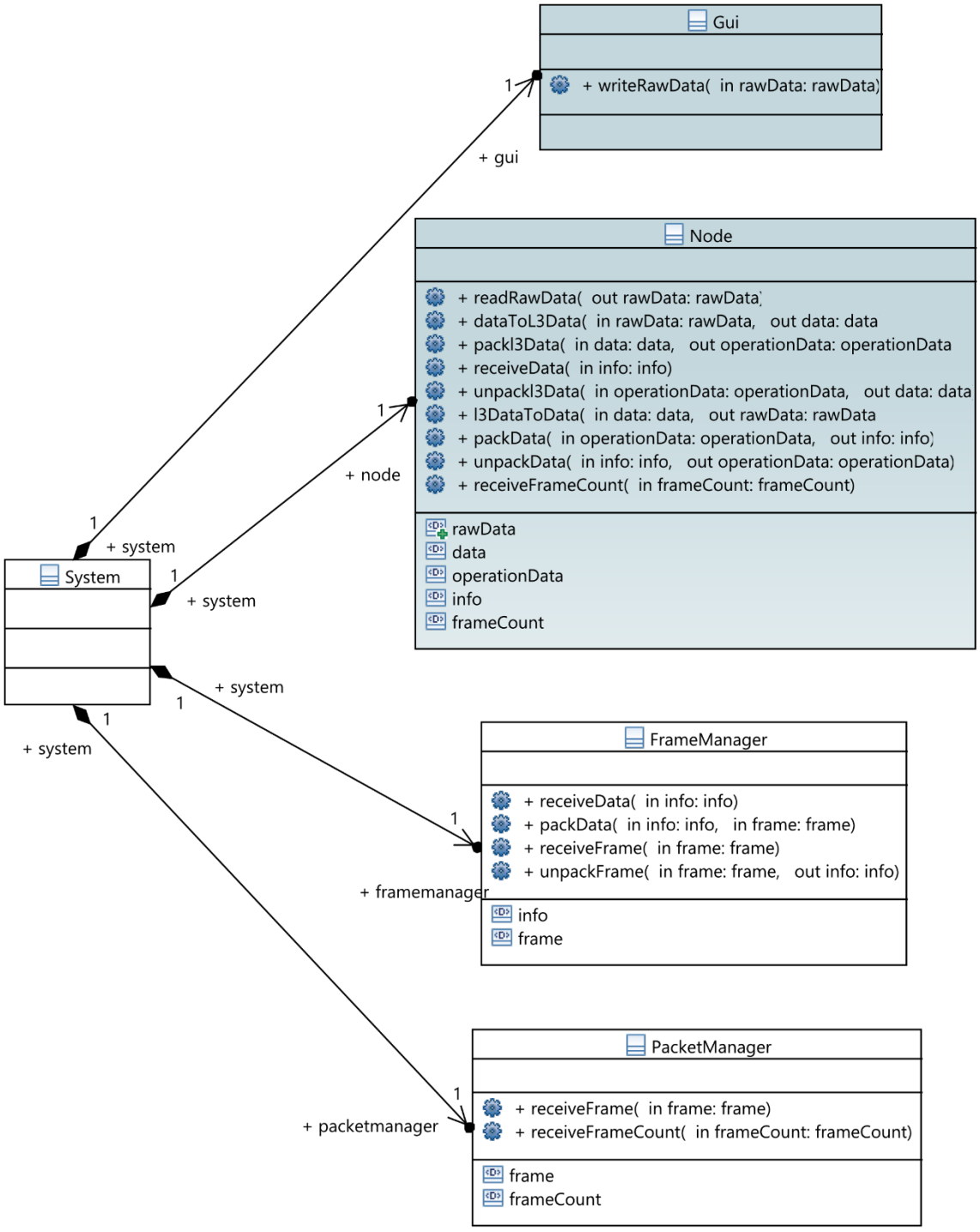


Figure 5. Class diagram of components with data types and procedure calls.

In terms of design, the model has been divided into four components: *Gui*, *Node*, *FrameManager* and *PacketManager*. These components are used to represent the different stages of a frame assembly. Each component has a sending and receiving state,

which contain the logic for sending or receiving the data. The components are designed so that they could be used both on the ground and on the satellite, although usage on satellite requires separation of the components.

Both of the class diagrams are used as a graphical representation of the communication protocol and do not represent the actual TASTE model.

4.2 Sequence diagrams

Communication protocol and its scenarios can be described as sequence diagrams, which are divided into ground and satellite segments.

The communication starts with a *Gui* component that represents a buffer and can be used for testing the model. From *Gui*, the initialised data is forwarded to *Node*, which manages the communication. The data is split into L3 data frames and the count of these frames is forwarded to *PacketManager*, which is the lowest component in the model. After that, the frame is packed into *Operation Data* field and *Info* object, which is forwarded to *FrameManager*. The *FrameManager* packs the *Info* object into an AX.25 frame. As the *PacketManager* has already received the frame count, it gathers all the frames and packs them into a packet, which represents the burst. The burst is then sent to the satellite. The same components are used when receiving the frames. The process is inverted and the received data is represented in *Gui*. As *Node* is responsible for communication management, it assembles the L3 data acknowledgement frames that contain the information about the frames that have not been received, so that they could be resent.

Specific communication scenarios are described in the following paragraphs.

4.2.1 Success

During a successful communication, the sender assembles the data into frames and forwards them as a burst. The burst is sent over TCP/IP (Transmission Control Protocol/Internet Protocol) protocol to the radio and converted into an analog signal. Receiver converts the signal back to a digital format and sends it over TCP/IP for disassembly. After the data has been received, an L3 data acknowledgement frame is assembled and sent back.

Sequence diagram representing the successful communication on ground is shown in Appendix 1 and on satellite is shown in Appendix 2. The connections between ground and satellite diagram are added as comments.

4.2.2 Partial success

If a failure occurs during the communication, as some frames have been lost or a timeout has occurred, the frames are sent or received partially. During a partially successful communication, the L3 acknowledgement frame is sent after reception which contains the missing frame numbers. Based on that, a burst can be assembled that contains the missing frames.

Sequence diagram representing the partially successful communication on ground is shown in Appendix 3 and on satellite is shown in Appendix 4. The connections between ground and satellite diagram are added as comments.

4.2.3 Timeout

As the communication window is limited, a timeout can occur which results in some frames getting lost. In that case, the sender initialises communication in the next communication window and requests for the frame numbers that were not received. Based on the L3 acknowledgement frame received, it can resend the missing frames.

Sequence diagram representing the timeout on ground is shown in Appendix 5 and on satellite is shown in Appendix 6. The connections between ground and satellite diagram are added as comments.

4.3 TASTE model

The TASTE model is using the data types that have been defined in the *data view*. The definition is based on the TTÜ-Mektory Nanosatellite TM/TC Protocol Description [17] and contains the data types of the frames and their fields. *Data view* also contains the data types that are used for describing additional parameters in the procedures.

Data types with descriptions in ASN.1 notation are shown in Appendix 7.

The TASTE model is divided into two parts: *interface view* and *deployment view*. *Interface view* is used for modeling software components with their logic and *deployment view* is used for defining software architecture.

In *interface view*, a four layer model was created in order to represent the components.

Based on SDL restrictions, each component is sporadic and contains asynchronous connections that are used for sending and receiving the data. As the components are universal, the model can be used both on the ground station and the satellite.

The components with connections in the *interface view* are shown in Figure 6.

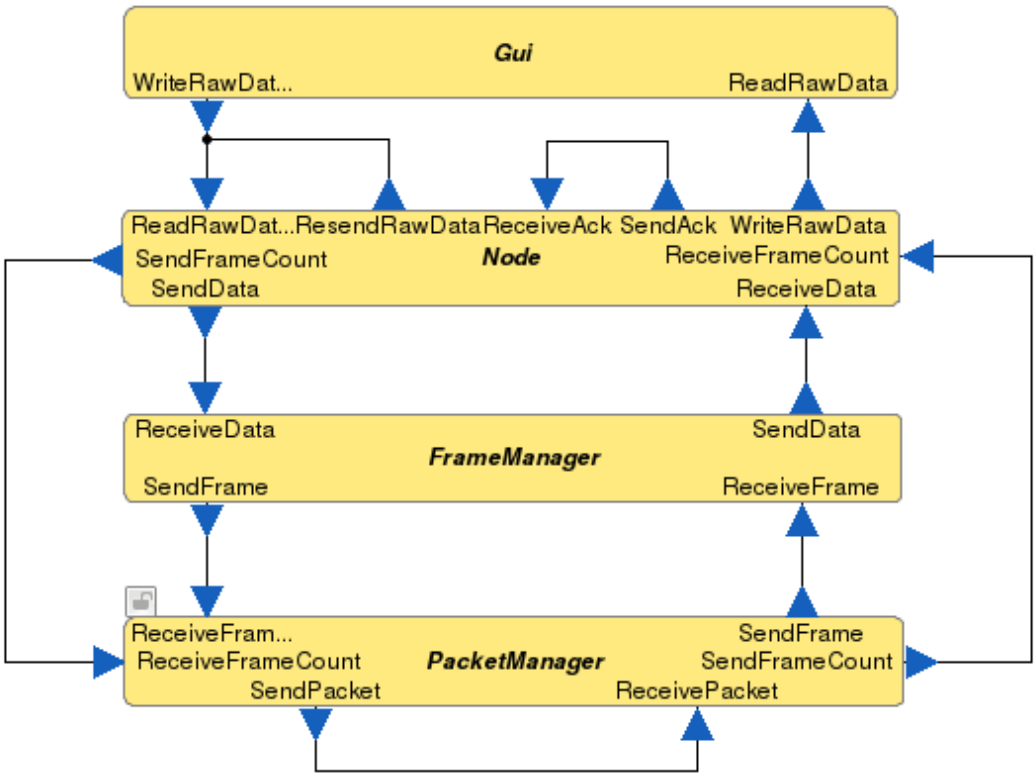


Figure 6. Components with connections in the interface view.

In *deployment view*, the architecture of the system has been described. The description of the architecture is shown in Figure 7.

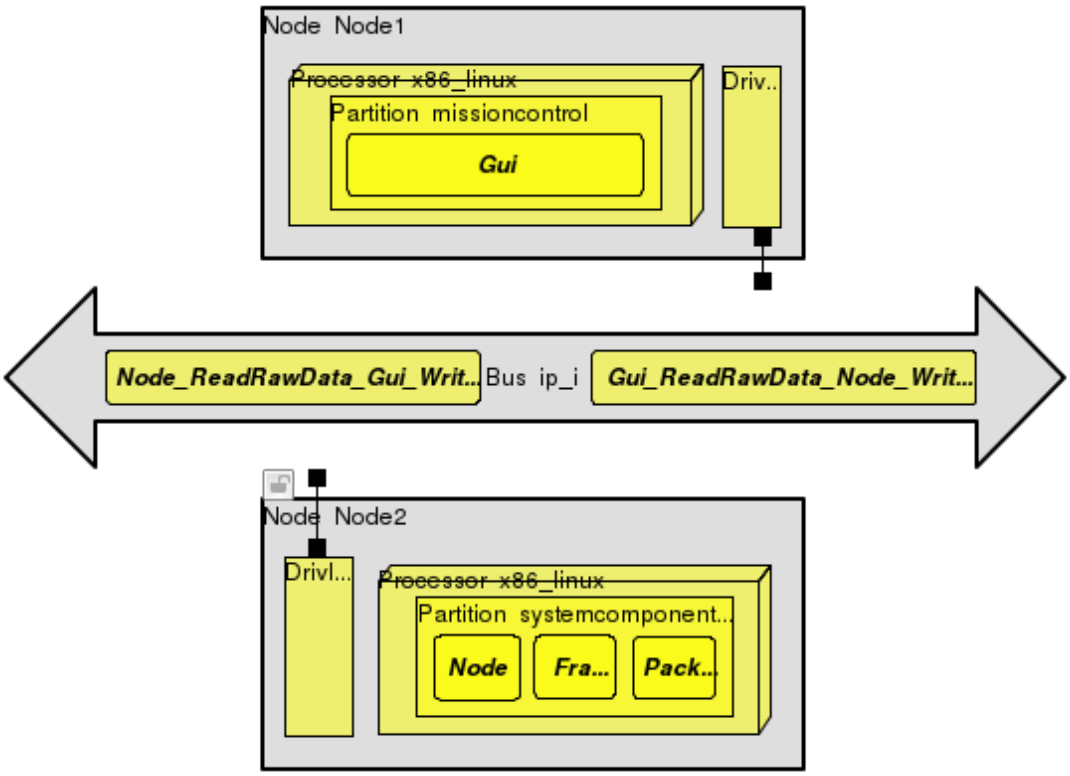


Figure 7. Architecture in deployment view.

The system consists of two nodes, which contain an x86 Linux processor with a partition. The partition of first node contains the *Gui* component and the partition of second node contains the three system components. Both of the nodes have drivers that are used for communication with the bus. The driver configuration consists of device name, address and port. As the system is using Linux sockets, the address is set to 127.0.0.1 which represents the local host and port values start from 5115. When testing the communication, another node could be connected to the bus to represent both the ground station and satellite.

Detailed description of each component is in the following paragraphs.

4.3.1 Gui

Gui is used for interacting with the *Node*. It is an interface that is used for testing the communication protocol. With *Gui*, raw data can be defined and sent to the *Node*. The data that is received from the *Node* can be used to validate the correctness of the communication protocol. *Gui* is a user-friendly interface for representing a buffer.

4.3.2 Node

Node is used for communication management both on the ground and on the satellite.

In the sending state it is used for parsing raw data. Raw data is received as an array of octets and it is split between L3 data objects, which contain *TtE*, *MRT* and *Fnum* fields. The count of L3 data objects is sent to *PacketManager*. L3 data objects are then packed into *Operation Data*, which is a part of *Info* object and accompanied with *BA*, *Cntrl*, *Code* and *IFCS* fields. *IFCS* is calculated based on the preceding fields according to the CRC [23, 24]. *Info* object, which is a part of AX.25 frame, is forwarded to the *FrameManager*. Besides frame assembly, *Node* is also responsible for stack management, so whenever a frame gets lost during the communication and an L3 acknowledgement frame with the lost frame numbers is received, it will resend these frames.

In the receiving state, *Node* unpacks the *Info* objects and *Operation Data*. It checks whether all the frames that were sent have been received. If some frames were not received or the checksum of the received frames was faulty, it assembles an L3 acknowledgement frame with the missing frame numbers. The frames that were received are sent to the *Gui*, which is a representation of a buffer. Whenever the received frame stack is overwritten for some reason, the received frames are not lost.

4.3.3 FrameManager

FrameManager receives the *Info* object from the *Node* or the frame from the *PacketManager*, depending on whether it is in the sending or receiving state.

When receiving an *Info* object from the *Node*, *FrameManager* assembles an AX.25 frame. In addition to the data specified in the *Info* object, the AX.25 frame also contains the *Addr*, *PID* and *FCS* fields. The *FCS* is calculated based on the preceding fields according to the CRC [23, 24]. If the AX.25 frame has been assembled, it is forwarded to the *PacketManager*.

When receiving the frame from the *PacketManager*, *FrameManager* unpacks the AX.25 frame and calculates the checksum. If the calculated checksum does not equal to the provided checksum, it does not forward the *Info* object to the *Node*. As the *Info* object is not forwarded, the *Node* will set the frame as missing.

4.3.4 PacketManager

PacketManager is used for burst handling, whether it is in the sending or receiving state.

In the sending state, the *PacketManager* receives the frame count from the *Node*. When all of the frames have been received, it packs the frames into a packet which represents the burst and sends it.

In the receiving state, the *PacketManager* receives the frames in a packet, counts the frames and sends the frame count to the *Node*. After the frame count has been sent, it sequentially sends the frames to *FrameManager*.

5 Code generation

After modeling the communication protocol with TASTE tools, the deployable code had to be generated.

The code skeletons for data types were already generated, as they were necessary for modeling the system. In order to generate deployable code that is based on system requirements, the SDL model had to be verified. The errors were eliminated and the warnings were fixed to avoid issues during runtime. Before generating usable C code, the Ada code for each component (*Node*, *FrameManager* and *PacketManager*) was generated. That resulted with an archive of each component, which contained the corresponding system logic.

Based on the component definitions and automatically generated build script, attempts were made to generate C code. During the generation, terminal gave multiple errors as an output. The errors were a result of the tool having inadequate support for specific use-cases, some of which were resolved in collaboration with engineers from ESA. The errors with references to commits in ESA OpenGEODE GitHub repository included:

- Issues with Unicode formatting on Ada back-end, fixed in commit 44a39ffedd3386ed68a7387d172e1e9da1492bc4 [25];
- Issues with Ada generator not being able to cast integers of similar data types, fixed in commit 4a1ca33e4fee2faa293427e26bc9c48bc621193e [26];
- Issues with Ada generator not being able to cast 64-bit integers from ASN.1 compiler to 32-bit integers, fixed in commit 1a60340a3a06518724f58b4ee1423573a01ce177 [27];
- Presence of shadow variables in for-loops and forbidden keywords in variable names, fixed in commits 90a7975742ec367eeebfbcb9445fd9592b513959 and 89e3a3e92fa725a9e11fa868258e57ef48bb5fa2 [28, 29].

- Issues parsing and generating code of the append function in OpenGEODE, not fixed as of 15.05.2017.

As a result of these issues, OpenGEODE was gradually updated from version 1.5.29 to version 1.5.32. In addition to that, parts of the TTÜ-Mektory Student Satellite communication protocol model were included in the test suite of OpenGEODE. The test cases were added in commit c0f40a86e6f0ff45d6e0fc473550875ddaca8f52 [30].

To avoid runtime issues with stack sizes and messages provoking stack overflow, the default stack size had to be increased. In TASTE Toolset, the default stack size for embedded systems has been configured to 50kB. As the variables used in the TTÜ-Mektory Student Satellite communication protocol were larger, the stack size of each thread was increased to 8192kB

After updating the TASTE Toolset and fixing issues in the model, the C code was generated. The binary files of the application with the GUI component were automatically placed inside a separate directory. In order to test the application, all of the binary files were executed inside a terminal.

6 Testing

TASTE Toolset had issues with code generation and testing, so the initial model was split into three separate components. In order to test the system logic of components, each of the components was connected to a separate GUI component. GUI components were used to initialise and send the required data to the system component. Depending on the component, the received data was packed and unpacked. To avoid issues with the append function in OpenGEODE, the system logic was simplified and single frames were used. The purpose of these tests was to validate the system logic of each component, before testing the message flow of the complete system.

The model for testing the *Node* component is shown in Figure 8.

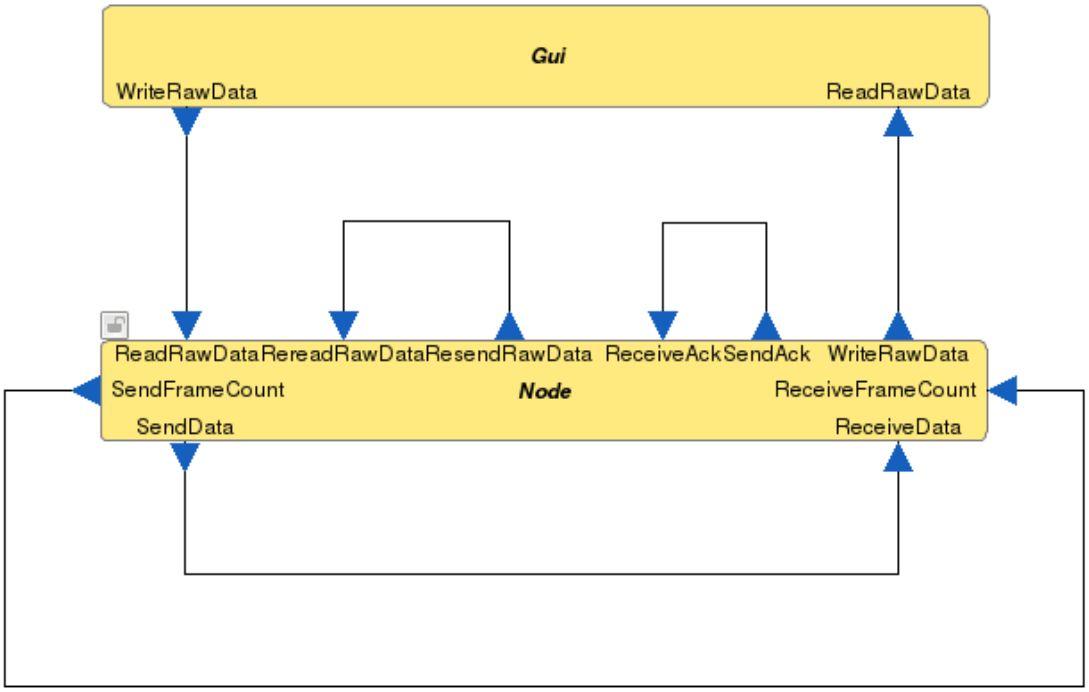


Figure 8. Node component for testing with connections in the interface view.

As the append function was not working, the test of the *Node* component was unsuccessful. The raw data was sent to the component after which it was split into L3 data, packed into *Info* object, sent, received and unpacked. After receiving the *Info* object, acknowledgement frame was assembled and sent. Although the *Info* object parameters

were received in GUI component, the raw data was not reinitialised from the received L3 data.

The model for testing the *FrameManager* component is shown in Figure 9.

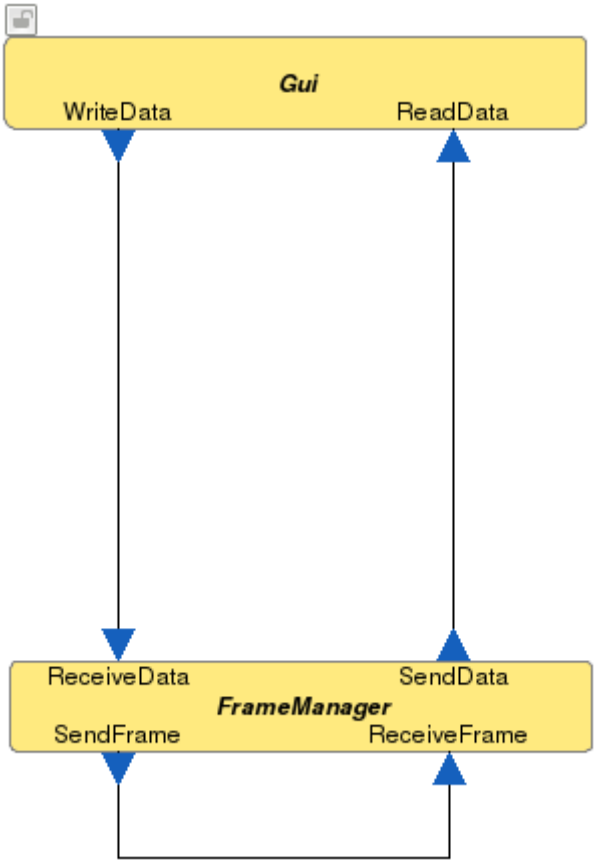


Figure 9. FrameManager component for testing with connections in the interface view.

The test of the *FrameManager* component was successful. *Info* object was sent to the component after which it was packed into an AX.25 frame, sent, received and unpacked. As a result the sent and received *Info* objects in GUI component were identical. The results of the test are shown in Figure 10 and Figure 11.

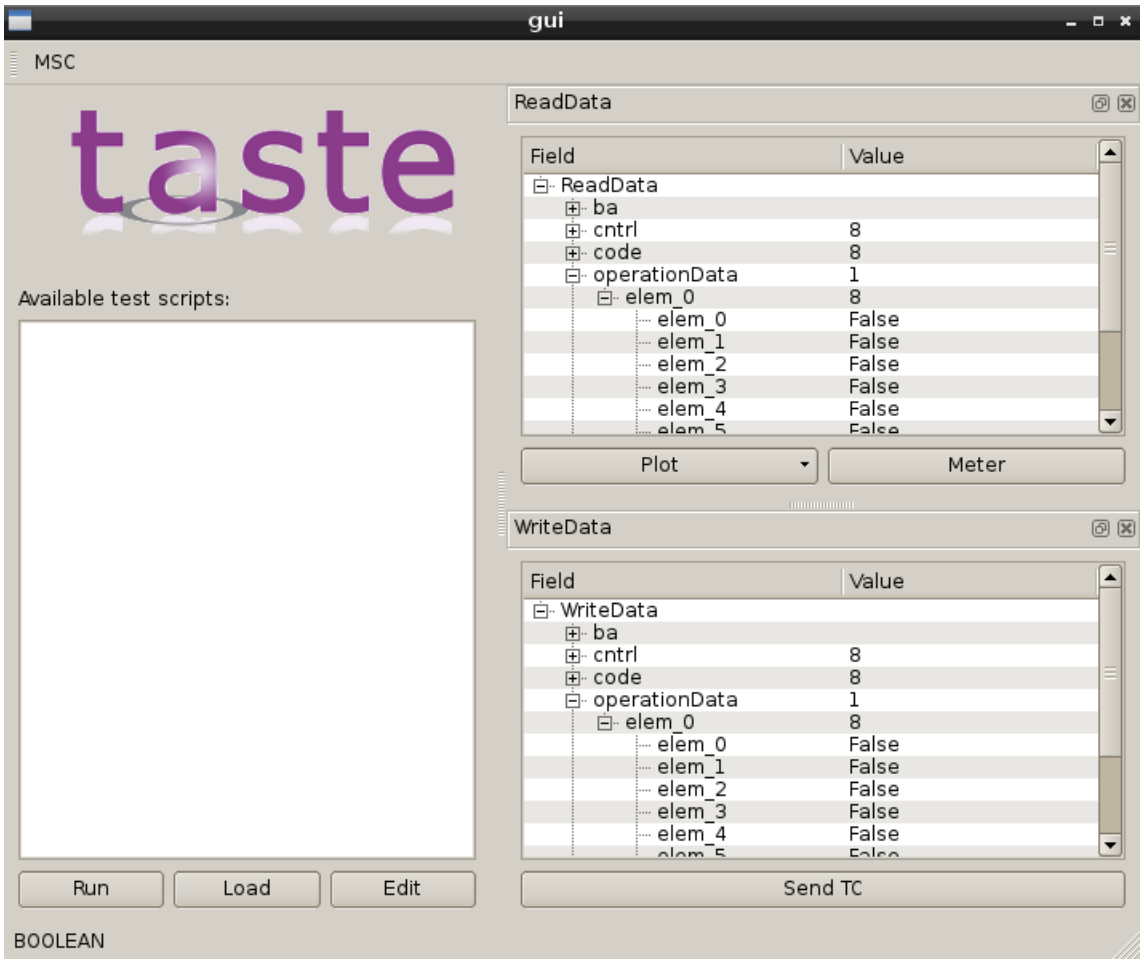


Figure 10. FrameManager component test written and read data.

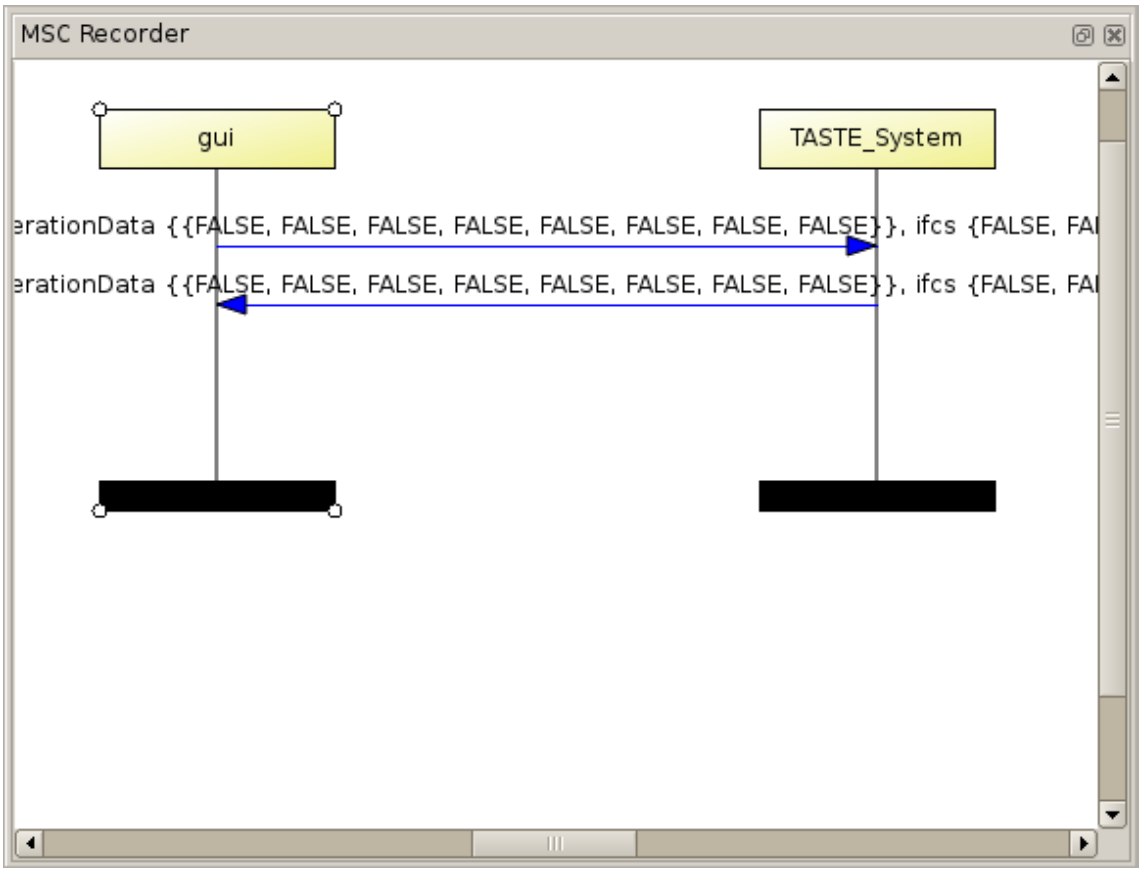


Figure 11. FrameManager component test graph.

The model for testing the *PacketManager* component is shown in Figure 12.

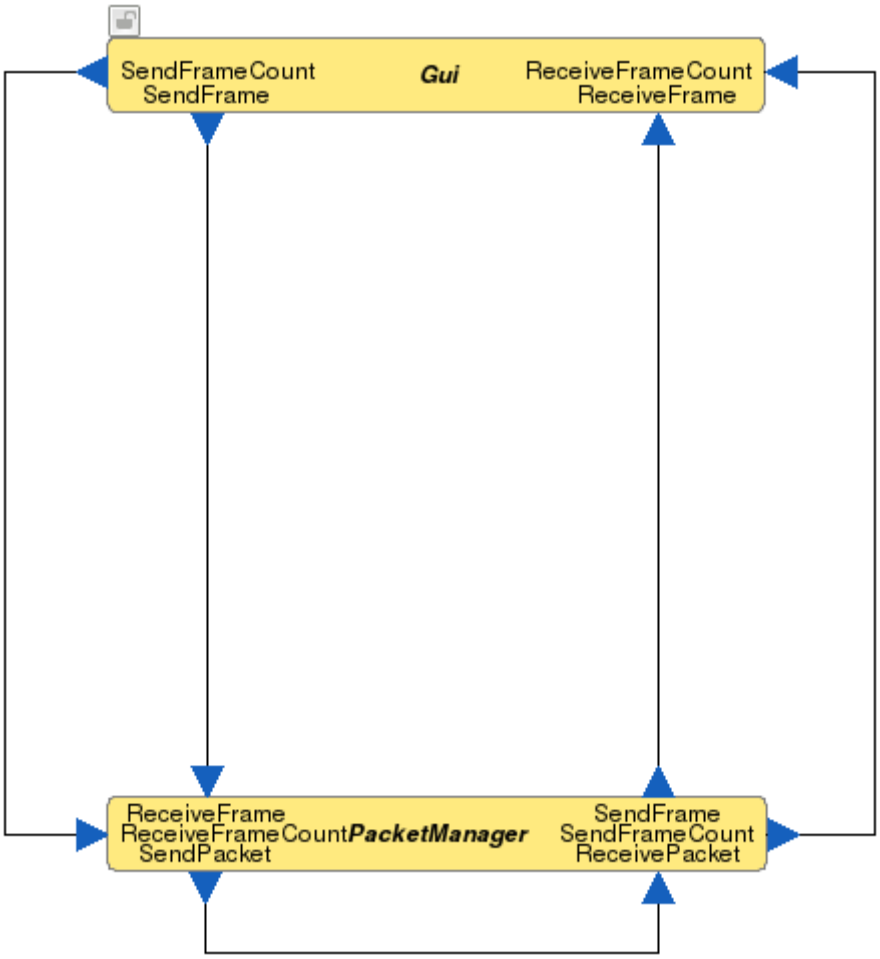


Figure 12. PacketManager component for testing with connections in the interface view.

The test of the *PacketManager* component was also successful. Frame count was sent to the component after which it changed its state and waited for AX.25 frames. After sending a number of AX.25 frames to the component a packet was packed, sent, received and unpacked. The frame count with the AX.25 frames was sent after receiving the packet. As a result the sent and received frame counts and AX.25 frames were identical. The results of the test are shown in Figure 13 and Figure 14.

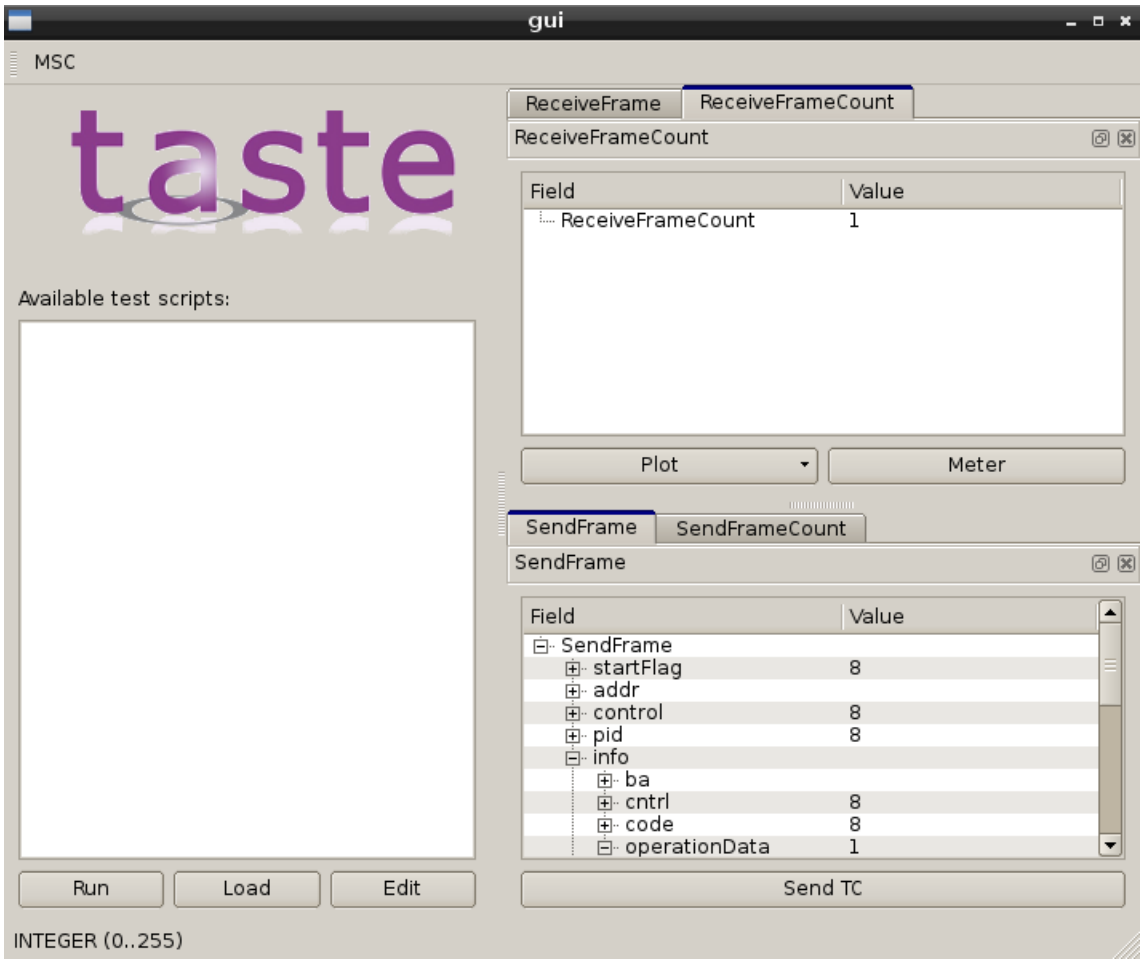


Figure 13. PacketManager component test written and read data.

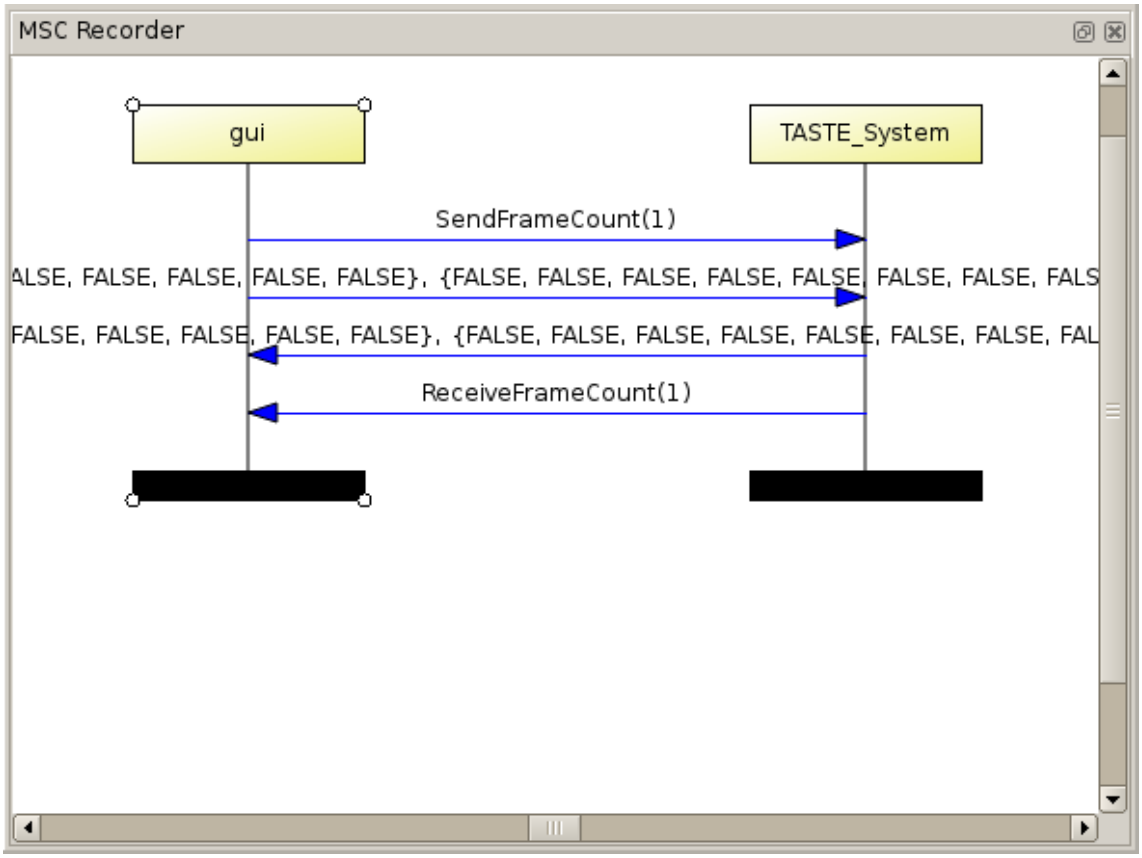


Figure 14. PacketManager component test graph.

Although the tests for separate components show that the system logic of the components is correct, some of the functionality could not be tested. In order to validate the system correctness, similar tests should be carried out for the complete system.

7 Results

In this thesis the communication protocol of TTÜ-Mektory Student Satellite was studied. The requirements of the system were met and the communication structure was described as a class diagram. The process was modeled as a sequence diagram and based on this model, some of the questions were answered and the communication protocol description was improved.

An SDL model containing the system logic was created in TASTE. The system architecture was described and attempts were made to generate usable C code. As the generation of C code had issues which required additional support from ESA engineers, the testing of the communication protocol was delayed. The issues were fixed and the TASTE Toolset was improved. As a result of this thesis, the TASTE Toolset has better validation in terms of model checking and now it supports some of the use-cases that had not been considered so far. In addition to that, exemplary functions can be created in the models that have been described in TASTE. Parts of the communication protocol were added to the test suite of OpenGEODE and the model will be used as a case study in ESA project that integrates TASTE Toolset with QGen.

After the issues had been solved, C code was generated and the system components were tested separately. As a result of this thesis, the communication protocol has been modeled and an implementation of the TTÜ-Mektory Student Satellite communication process has been created. The binary files containing the C code of the system can be validated for correctness and deployed on the target system.

8 Future work

As the testing of the communication protocol implementation was delayed, the generated C code was tested partially. Before deploying the C code on the target system, the system logic should be verified completely.

The verification can be done in the TASTE Toolset, which has different tools for validation. The binary files could be run and tested against the GUI component, which gives an overview of the system logic. After the initial verification, the ground and space segment could be initialised. As the system was modeled so that it could be used both on ground and on the satellite, another instance of the model could be created. The architecture of the ground station and satellite could be described and the whole communication process could be tested. For example the GUI component could give raw data as an input to the ground station and receive the contents of an acknowledgement frame from the satellite. After testing the communication process, different communication scenarios could also be verified. These scenarios include successful communication, partially successful communication and timeout.

As the bus protocol commands were not considered as a part of this thesis, the frame structure of these commands could be described. The logic for sending and receiving these commands could be modeled and used by the system in case the OBC of the satellite is not accessible.

After the system has been modeled as a whole and verified, the code could be deployed as a separate component in the ground station of the TTÜ-Mektory Student Satellite. Parts of the system could also be used on the satellite, although that requires modifications to the system logic and separation of the component architecture.

9 Summary

The purpose of this thesis was to use TASTE Toolset for modeling the communication protocol of TTÜ-Mektory Student Satellite and generate C code based on that model.

In this thesis, a part of the TTÜ-Mektory Student Satellite communication was modeled. That part consisted of ground and space segment, which were used for communication between ground station and satellite on-board computer. The system components were designed so that they could be used both on the ground and on the satellite.

The description of TTÜ-Mektory Student Satellite communication protocol was studied and based on that description, the system was described. Class diagrams describing the communication structure and sequence diagrams describing the communication process were created. The system was modeled in TASTE Toolset, where data types were described in ASN.1 and components of the system were modeled in SDL. Ada code for the system components was generated and the system architecture was described. Based on that description, C code was generated.

The code generation resulted with errors, which were caused by the lack of support for some of the use-cases. The errors were resolved in collaboration with engineers from ESA and as a result the TASTE Toolset was improved. Parts of the TTÜ-Mektory Student Satellite communication protocol were added to the test suite of the toolset.

Although the C code was generated, the errors in code generation delayed the testing of the communication protocol. Based on the current state, the description of communication protocol is sufficient for modeling the software system. As the communication protocol was not thoroughly tested, it is difficult to evaluate whether the requirements are sufficient for describing the communication process. The generated code is usable in the TTÜ-Mektory satellite program both on ground and on the satellite, although the tests for the complete system will indicate whether the model requires any improvements. As the TASTE Toolset is used for modeling software systems that are used in real-time environments, it is also sensible to use it for modeling the other software components for TTÜ-Mektory satellite program. Although the TASTE Toolset may require some

improvements, it keeps consistency throughout the whole system and ensures that it is actually based on the requirements.

Finally, I would like to thank my supervisor Evelin Halling and the engineers from IB Krates and ESA.

References

[1] TASTE, “TASTE,” [Online]. Available: http://taste.tuxfamily.org/. [Accessed 30 04 2017].

[2] M. Perrotin, E. Conquet, J. Delange and T. Tsiodras, “TASTE: An open-source tool-chain for embedded system and software development,” 2012.

[3] M. Perrotin, K. Grochowski, M. Verhoef, D. Galano, M. Mosdorf, M. Kurowski, F. Denis and E. Graas, “TASTE in action,” 2016.

[4] M. Perrotin, E. Conquet, P. Dissaux, T. Tsiodras and J. Hugues, “The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software,” 2010.

[5] TASTE, “Overview,” [Online]. Available: http://taste.tuxfamily.org/wiki/index.php?title=Overview. [Accessed 22 03 2017].

[6] TASTE, “QGen,” [Online]. Available: http://taste.tuxfamily.org/wiki/index.php?title=QGen. [Accessed 22 03 2017].

[7] ITU-T, Specification and Description Language – Overview of SDL-2010, 2016.

[8] TASTE, “Technical topic: OpenGEODE, an SDL editor for TASTE,” [Online]. Available: http://taste.tuxfamily.org/wiki/index.php?title=Technical_topic:_OpenGEODE,_an_SDL_editor_for_TASTE. [Accessed 22 03 2017].

[9] G. Mamais, T. Tsiodras, D. Lesens and M. Perrotin, “An ASN.1 compiler for embedded/space systems,” 2012.

[10] TASTE, “Technical topic: ASN1SCC - ESA's ASN.1 Compiler for safety-critical embedded platforms,” [Online]. Available: http://taste.tuxfamily.org/wiki/index.php?title=Technical_topic:_ASN1SCC_-_ESA%27s_ASN.1_Compiler_for_safety-critical_embedded_platforms. [Accessed 22 03 2017].

[11] TASTE, “ASN.1 generators,” [Online]. Available: http://taste.tuxfamily.org/wiki/index.php?title=ASN.1_generators. [Accessed 22 03 2017].

[12] J. Hugues, L. Pautet, P. Dissaux and M. Perrotin, “Using AADL to build critical real-time systems: Experiments in the IST-ASSERT project,” 2008.

[13] TASTE, “pus-asn,” [Online]. Available: http://download.tuxfamily.org/taste/ASN1SCC/pus-asn.zip. [Accessed 22 03 2017].

[14] TASTE, “pus-taste,” [Online]. Available: http://download.tuxfamily.org/taste/ASN1SCC/pus-taste.tar.gz. [Accessed 22 03 2017].

[15] P.-T. G. Estévez, UPMSat-2 satellite’s Manager software subsystem: Design, validation, implementation and verification - UPM, 2014.

[16] M. Perrotin, E. Conquet, J. Delange, A. Schiele and T. Tsiodras, “TASTE: A real-time software engineering tool-chain. Overview, status, and future,” 2011.

[17] R. Adelbert, TTÜ-Mektory Nanosatellite: Satellite TCTM Protocol Description, 2016.

[18] W. A. Beech, D. E. Nielsen and J. Taylor, AX.25 Link Access Protocol for Amateur Packet Radio, 1998.

[19] Eclipse, “Papyrus Modeling Environment,” [Online]. Available: https://eclipse.org/papyrus/. [Accessed 30 04 2017].

[20] TASTE, “Virtual Machine,” [Online]. Available: http://taste.tuxfamily.org/wiki/index.php?title=Virtual_Machine. [Accessed 01 05 2017].

[21] Oracle, “VirtualBox,” [Online]. Available: https://www.virtualbox.org/. [Accessed 01 05 2017].

[22] TASTE, “TASTE V2 Reference Card,” [Online]. Available: http://taste.tuxfamily.org/wiki/images/d/d1/Taste_refcard.pdf. [Accessed 01 05 2017].

[23] Wikipedia, “Cyclic redundancy check,” [Online]. Available: https://en.wikipedia.org/wiki/Cyclic_redundancy_check. [Accessed 11 04 2017].

[24] PracticingElectronics, “The Cyclic Redundancy Check (CRC) for AX.25,” [Online]. Available: http://practicingelectronics.com/articles/article-100003/article.php. [Accessed 11 04 2017].

[25] GitHub, “Ada backend: fix unicode issues,” [Online]. Available: https://github.com/esa/opengeode/commit/44a39ffedd3386ed68a7387d172e1e9da1492bc4. [Accessed 29 04 2017].

[26] GitHub, “Symetric cast in Equality tests,” [Online]. Available: https://github.com/esa/opengeode/commit/4a1ca33e4fee2faa293427e26bc9c48bc621193e. [Accessed 29 04 2017].

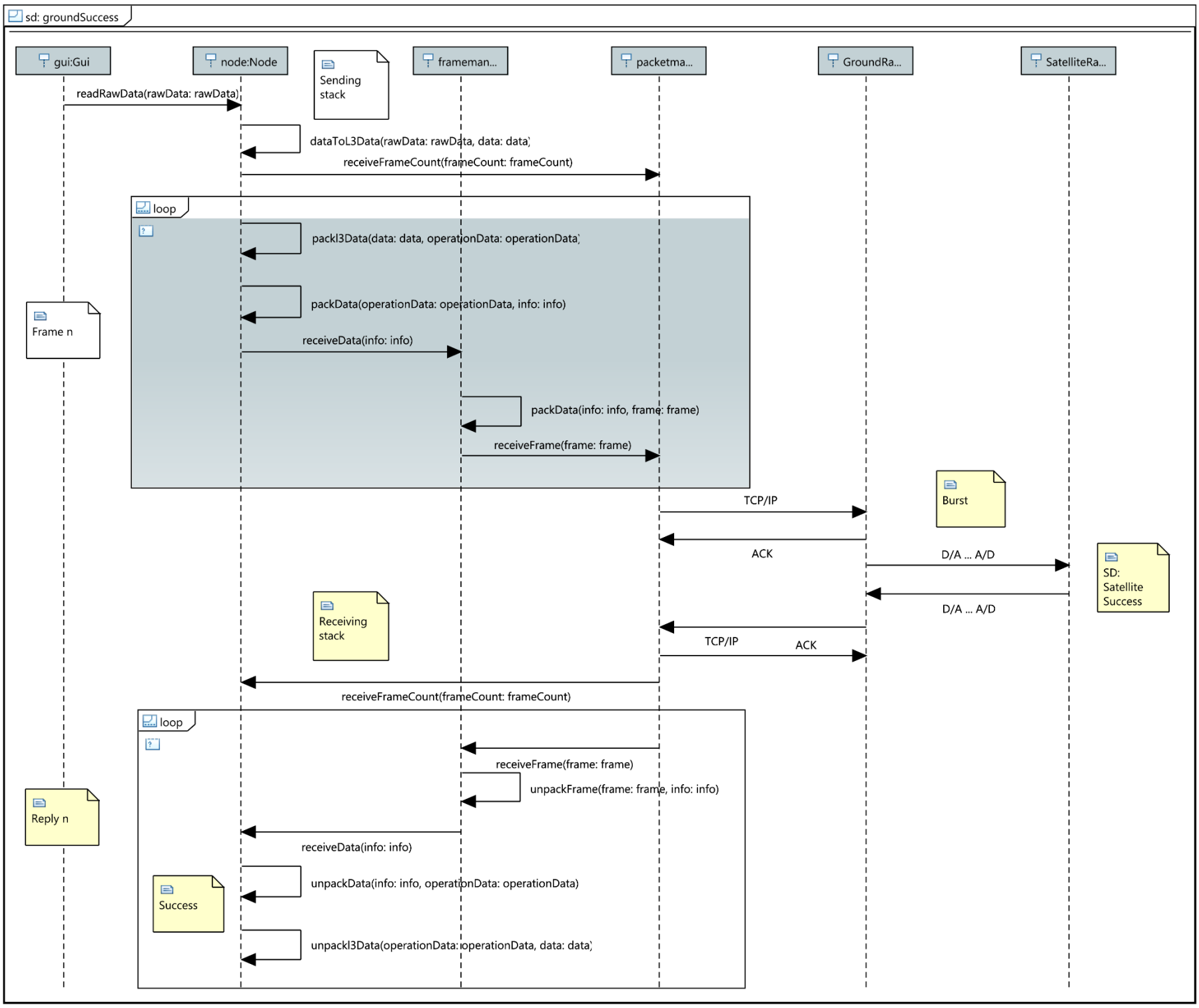
[27] GitHub, “Fix various 32bits/64bits conversion issues,” [Online]. Available: https://github.com/esa/opengeode/commit/1a60340a3a06518724f58b4ee1423573a01ce177. [Accessed 29 04 2017].

[28] GitHub, “Report shadow variables in for loops,” [Online]. Available: https://github.com/esa/opengeode/commit/90a7975742ec367eeebfbcb9445fd9592b513959. [Accessed 29 04 2017].

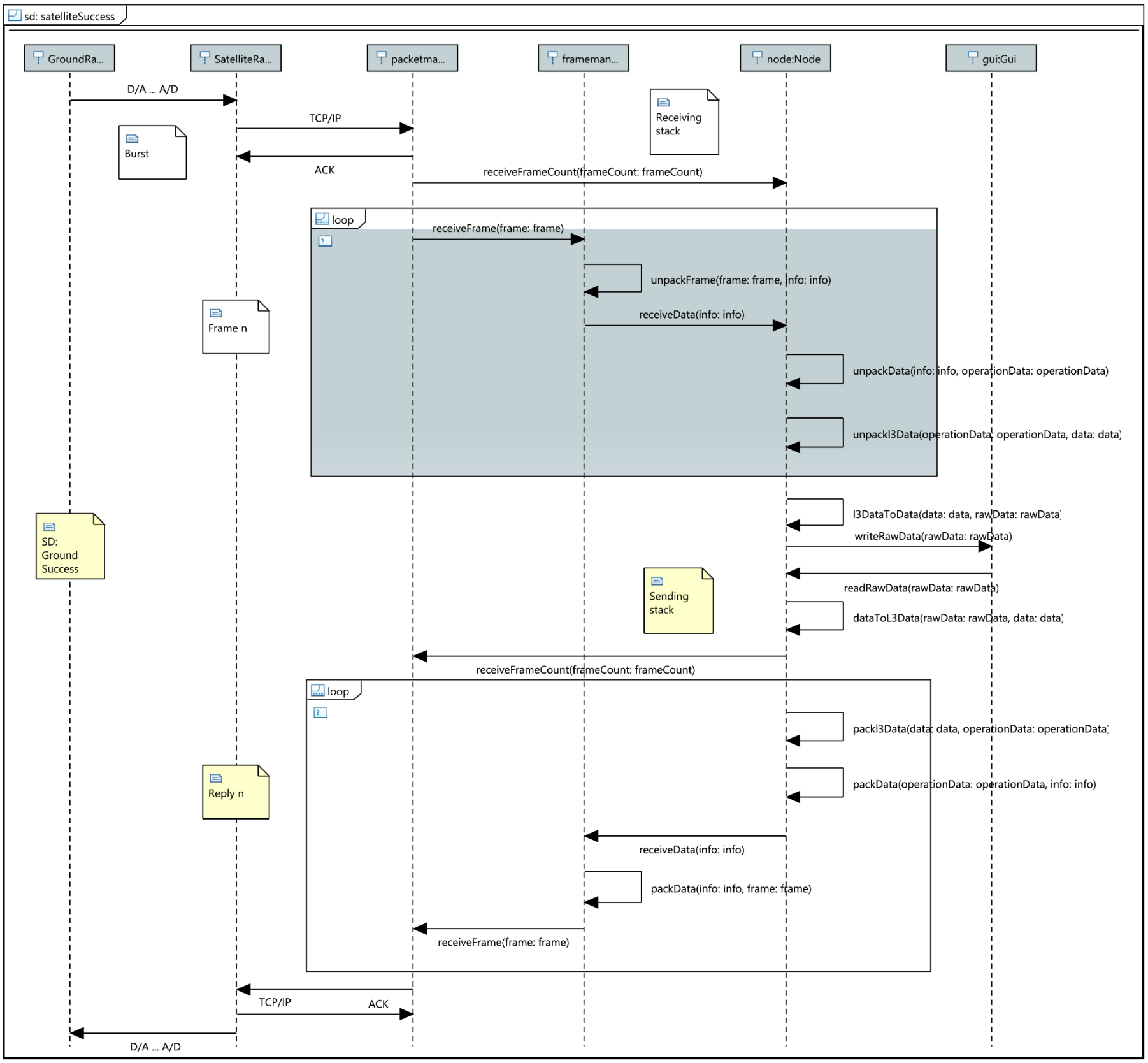
[29] GitHub, “Detect some forbidden keywords in variable names,” [Online]. Available: https://github.com/esa/opengeode/commit/89e3a3e92fa725a9e11fa868258e57ef48bb5fa2. [Accessed 29 04 2017].

[30] GitHub, “Add test cases from ib krates,” [Online]. Available: https://github.com/esa/opengeode/commit/c0f40a86e6f0ff45d6e0fc473550875ddaca8f52. [Accessed 29 04 2017].

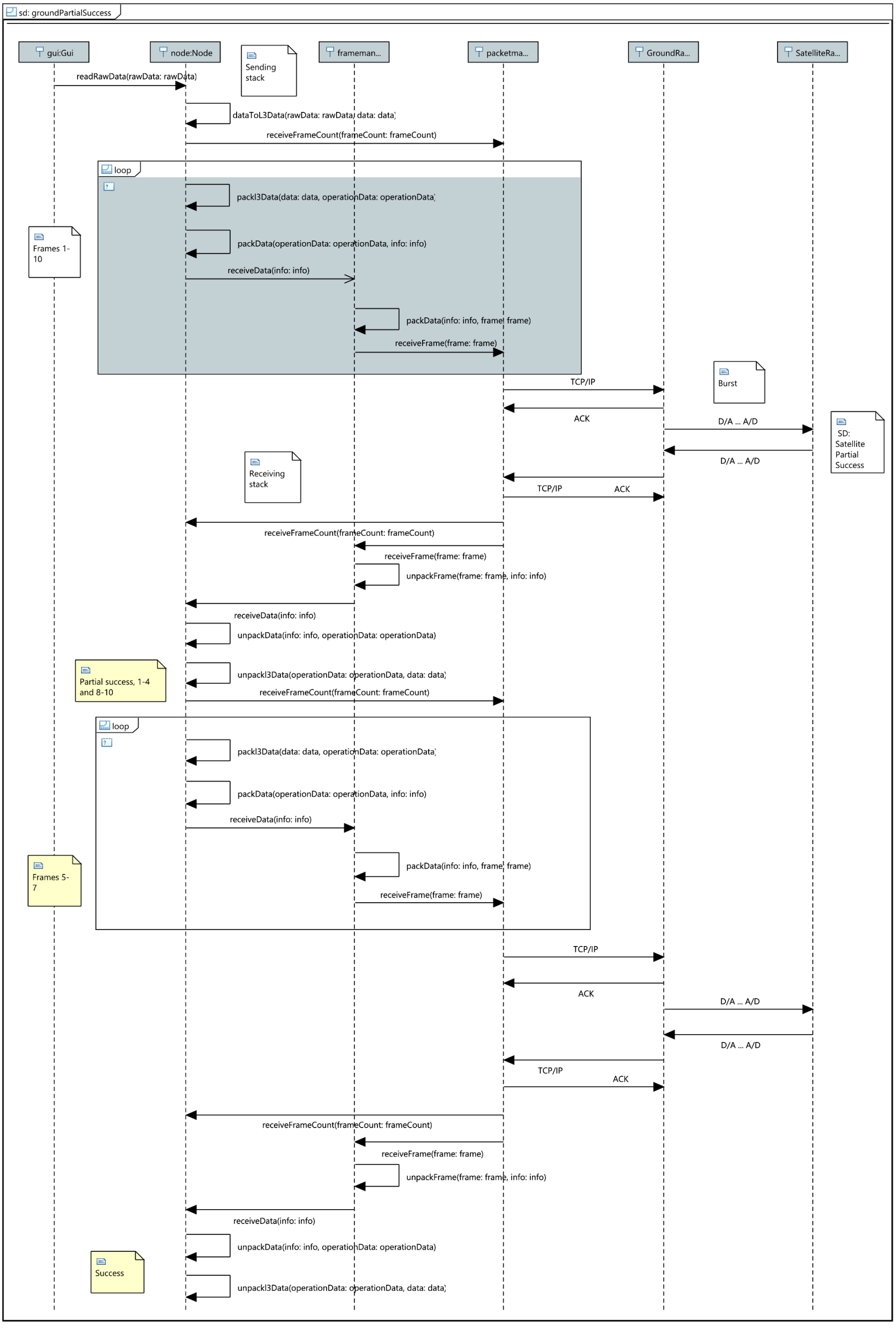
Appendix 1 – Sequence diagram of successful communication on ground


sd: groundSuccess
gui:Gui
node:Node
frameman...
packetma...
GroundRa...
SatelliteRa...
Sending stack
readRawData(rawData: rawData)
dataToL3Data(rawData: rawData, data: data)
receiveFrameCount(frameCount: frameCount)
loop
packl3Data(data: data, operationData: operationData)
packData(operationData: operationData, info: info)
receiveData(info: info)
packData(info: info, frame: frame)
receiveFrame(frame: frame)
Frame n
TCP/IP
Burst
ACK
D/A ... A/D
SD: Satellite Success
D/A ... A/D
Receiving stack
TCP/IP
ACK
receiveFrameCount(frameCount: frameCount)
loop
receiveFrame(frame: frame)
unpackFrame(frame: frame, info: info)
Reply n
receiveData(info: info)
Success
unpackData(info: info, operationData: operationData)
unpackl3Data(operationData: operationData, data: data)

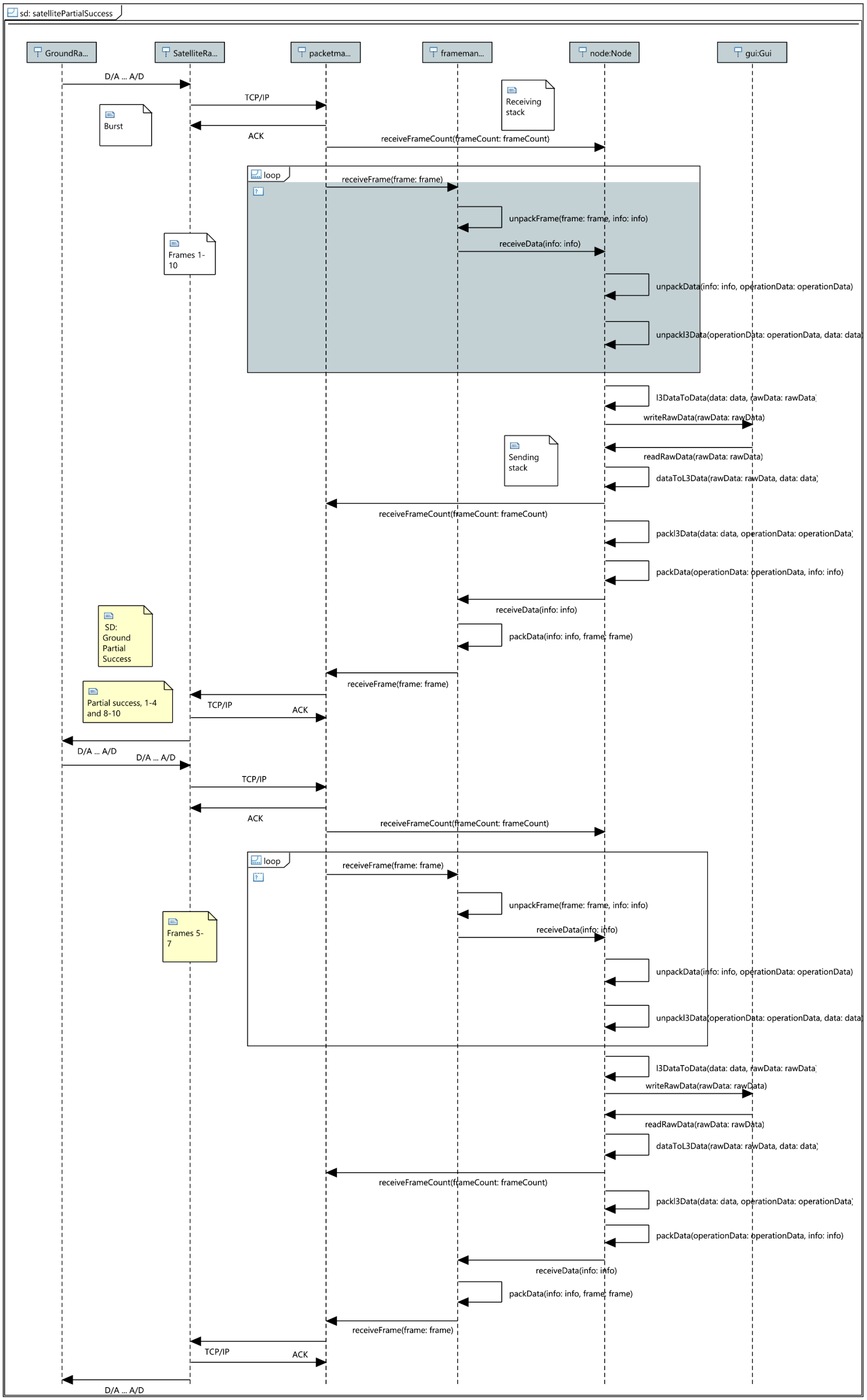
Appendix 2 – Sequence diagram of successful communication on satellite


sd: satelliteSuccess
GroundRa...
SatelliteRa...
packetma...
frameman...
node:Node
gui:Gui
D/A ... A/D
TCP/IP
Burst
ACK
Receiving stack
receiveFrameCount(frameCount: frameCount)
loop
receiveFrame(frame: frame)
unpackFrame(frame: frame, info: info)
Frame n
receiveData(info: info)
unpackData(info: info, operationData: operationData)
unpackl3Data(operationData: operationData, data: data)
l3DataToData(data: data, rawData: rawData)
writeRawData(rawData: rawData)
SD: Ground Success
readRawData(rawData: rawData)
Sending stack
dataToL3Data(rawData: rawData, data: data)
receiveFrameCount(frameCount: frameCount)
loop
packl3Data(data: data, operationData: operationData)
Reply n
packData(operationData: operationData, info: info)
receiveData(info: info)
packData(info: info, frame: frame)
receiveFrame(frame: frame)
TCP/IP
ACK
D/A ... A/D

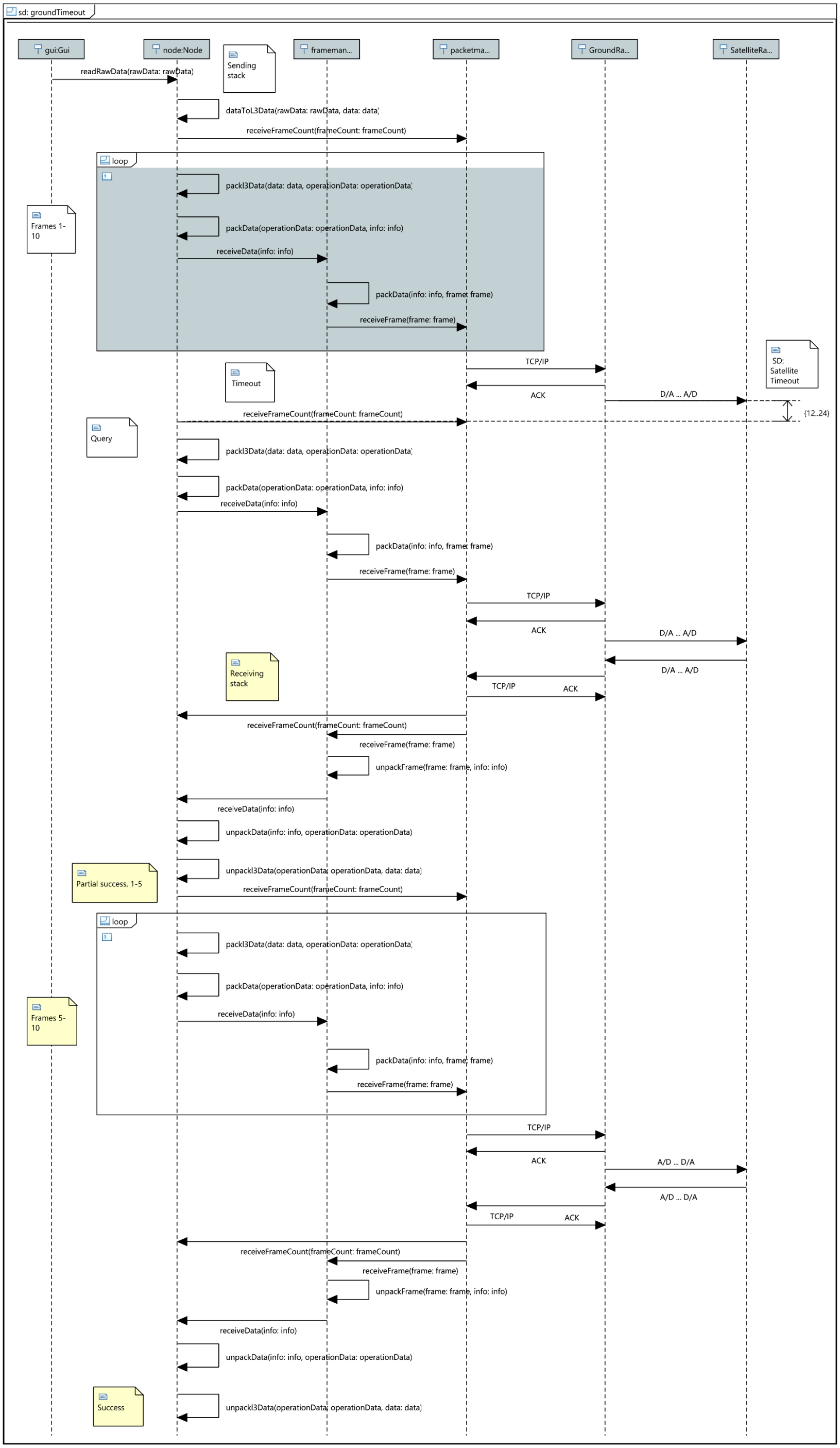
Appendix 3 – Sequence diagram of partially successful communication on ground


sd: groundPartialSuccess
gui:Gui
node:Node
frameman...
packetma...
GroundRa...
SatelliteRa...
readRawData(rawData: rawData)
Sending stack
dataToL3Data(rawData: rawData, data: data)
receiveFrameCount(frameCount: frameCount)
loop
packl3Data(data: data, operationData: operationData)
packData(operationData: operationData, info: info)
receiveData(info: info)
packData(info: info, frame: frame)
receiveFrame(frame: frame)
Frames 1-10
TCP/IP
Burst
ACK
D/A ... A/D
SD: Satellite Partial Success
D/A ... A/D
Receiving stack
TCP/IP
ACK
receiveFrameCount(frameCount: frameCount)
receiveFrame(frame: frame)
unpackFrame(frame: frame, info: info)
receiveData(info: info)
unpackData(info: info, operationData: operationData)
Partial success, 1-4 and 8-10
unpackl3Data(operationData: operationData, data: data)
receiveFrameCount(frameCount: frameCount)
loop
packl3Data(data: data, operationData: operationData)
packData(operationData: operationData, info: info)
receiveData(info: info)
packData(info: info, frame: frame)
receiveFrame(frame: frame)
Frames 5-7
TCP/IP
ACK
D/A ... A/D
D/A ... A/D
TCP/IP
ACK
receiveFrameCount(frameCount: frameCount)
receiveFrame(frame: frame)
unpackFrame(frame: frame, info: info)
receiveData(info: info)
unpackData(info: info, operationData: operationData)
Success
unpackl3Data(operationData: operationData, data: data)

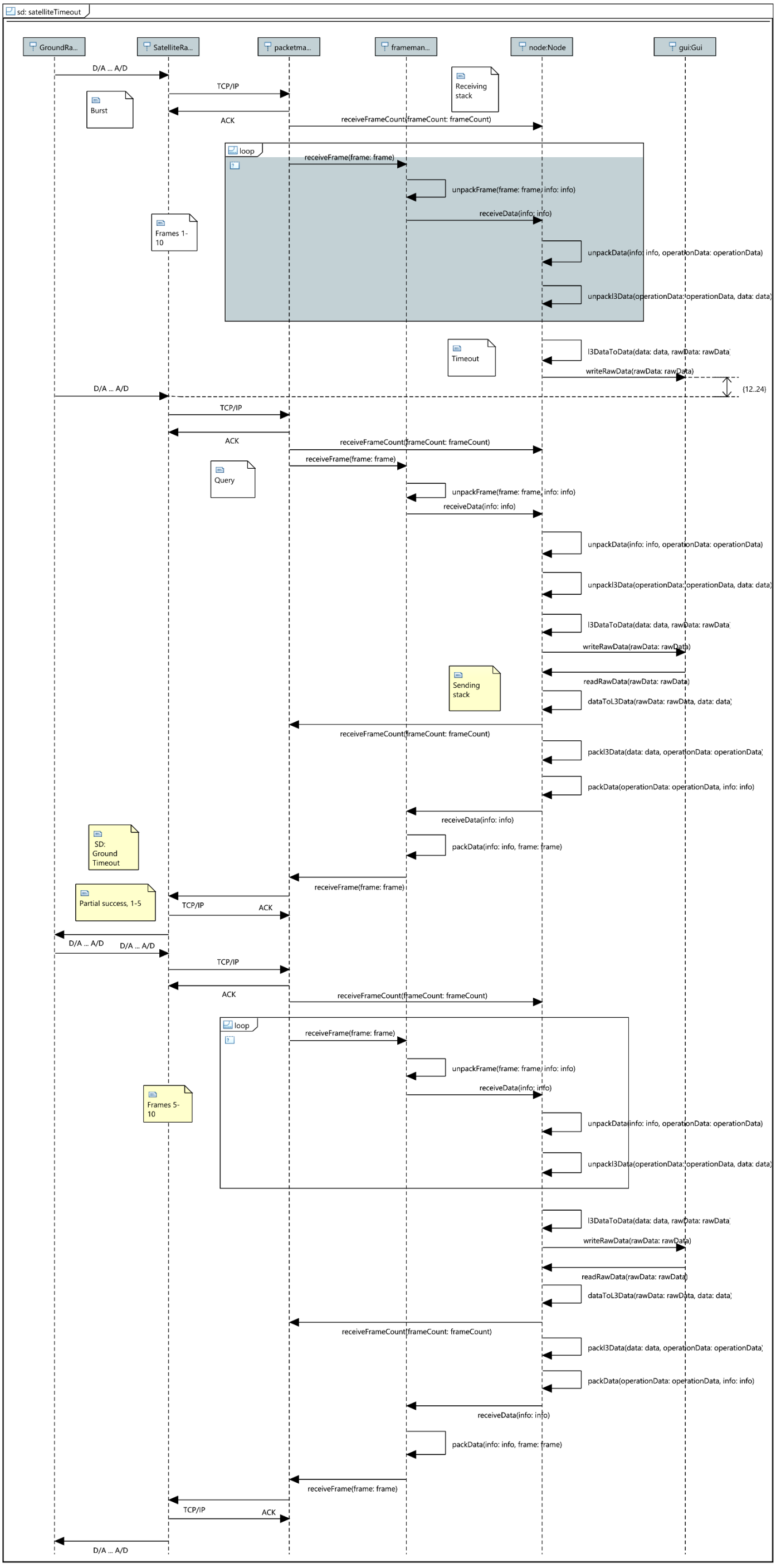
Appendix 4 – Sequence diagram of partially successful communication on satellite


sd: satellitePartialSuccess
GroundRa...
SatelliteRa...
packetma...
frameman...
node:Node
gui:Gui
D/A ... A/D
TCP/IP
Burst
ACK
Receiving stack
receiveFrameCount(frameCount: frameCount)
loop
receiveFrame(frame: frame)
unpackFrame(frame: frame, info: info)
receiveData(info: info)
Frames 1-10
unpackData(info: info, operationData: operationData)
unpackI3Data(operationData: operationData, data: data)
I3DataToData(data: data, rawData: rawData)
writeRawData(rawData: rawData)
Sending stack
readRawData(rawData: rawData)
dataToL3Data(rawData: rawData, data: data)
receiveFrameCount(frameCount: frameCount)
packI3Data(data: data, operationData: operationData)
packData(operationData: operationData, info: info)
receiveData(info: info)
SD: Ground Partial Success
packData(info: info, frame: frame)
receiveFrame(frame: frame)
Partial success, 1-4 and 8-10
TCP/IP
ACK
D/A ... A/D
D/A ... A/D
TCP/IP
ACK
receiveFrameCount(frameCount: frameCount)
loop
receiveFrame(frame: frame)
unpackFrame(frame: frame, info: info)
receiveData(info: info)
Frames 5-7
unpackData(info: info, operationData: operationData)
unpackI3Data(operationData: operationData, data: data)
I3DataToData(data: data, rawData: rawData)
writeRawData(rawData: rawData)
readRawData(rawData: rawData)
dataToL3Data(rawData: rawData, data: data)
receiveFrameCount(frameCount: frameCount)
packI3Data(data: data, operationData: operationData)
packData(operationData: operationData, info: info)
receiveData(info: info)
packData(info: info, frame: frame)
receiveFrame(frame: frame)
TCP/IP
ACK
D/A ... A/D

Appendix 5 – Sequence diagram of timeout on ground


sd: groundTimeout
gui:Gui
node:Node
frameman...
packetma...
GroundRa...
SatelliteRa...
Sending stack
readRawData(rawData: rawData)
dataToL3Data(rawData: rawData, data: data)
receiveFrameCount(frameCount: frameCount)
loop
packl3Data(data: data, operationData: operationData)
packData(operationData: operationData, info: info)
receiveData(info: info)
packData(info: info, frame: frame)
receiveFrame(frame: frame)
Frames 1-10
TCP/IP
ACK
D/A ... A/D
SD: Satellite Timeout
Timeout
{12..24}
Query
Receiving stack
Partial success, 1-5
unpackFrame(frame: frame, info: info)
unpackData(info: info, operationData: operationData)
unpackl3Data(operationData: operationData, data: data)
Frames 5-10
A/D ... D/A
Success

Appendix 6 – Sequence diagram of timeout on satellite


sd: satelliteTimeout
GroundRa...
SatelliteRa...
packetma...
frameman...
node:Node
gui:Gui
D/A ... A/D
TCP/IP
ACK
Burst
Receiving stack
receiveFrameCount(frameCount: frameCount)
loop
receiveFrame(frame: frame)
unpackFrame(frame: frame, info: info)
receiveData(info: info)
Frames 1-10
unpackData(info: info, operationData: operationData)
unpackl3Data(operationData: operationData, data: data)
Timeout
l3DataToData(data: data, rawData: rawData)
writeRawData(rawData: rawData)
{12..24}
Query
Sending stack
readRawData(rawData: rawData)
dataToL3Data(rawData: rawData, data: data)
packl3Data(data: data, operationData: operationData)
packData(operationData: operationData, info: info)
packData(info: info, frame: frame)
SD: Ground Timeout
Partial success, 1-5
Frames 5-10

Appendix 7 – Data types with descriptions in ASN.1 notation

```
TASTE-Dataview DEFINITIONS ::=
BEGIN

-- Boolean
MyBoolean ::= BOOLEAN

-- 4 bit array
My4BitArray ::= SEQUENCE (SIZE (4)) OF BOOLEAN

-- 8 bit array
My8BitArray ::= SEQUENCE (SIZE (8)) OF BOOLEAN

-- Integer for 8 bit array numbering
My8BitArrayInteger ::= INTEGER (0..7)

-- Integer for 8 bit array value numbering
My8BitArrayValueInteger ::= INTEGER (0..255)

-- 16 bit array
My16BitArray ::= SEQUENCE (SIZE (16)) OF BOOLEAN

-- Integer for 16 bit array numbering
My16BitArrayInteger ::= INTEGER (0..15)

-- Integer for 16 bit array value numbering
My16BitArrayValueInteger ::= INTEGER (0..65535)

-- Packet structure
MyPacket ::= SEQUENCE (SIZE (0..255)) OF MyFrame

-- Stack for data in bits
MyDataStack ::= SEQUENCE (SIZE (40..2000)) OF BOOLEAN

-- Integer for data stack value numbering
MyDataStackValueInteger ::= INTEGER (0..2000)

-- Stack for data of FCS calculation
MyFcsStack ::= SEQUENCE (SIZE (224..2192)) OF BOOLEAN -- Frame has empty info
or is full

-- Integer for data of FCS calculation
MyFcsStackInteger ::= INTEGER (0..2175)
```

```
-- Polynomial for FCS calculation
MyFcsPolynomial ::= SEQUENCE (SIZE (17)) OF BOOLEAN

-- Integer for polynomial numbering
MyFcsPolynomialInteger ::= INTEGER (0..16)

-- Frame structure
MyFrame ::= SEQUENCE {
  startFlag MyFlag,
  addr MyAddr,
  control MyControl,
  pid MyPid,
  info MyInfo,
  fcs MyFcs,
  endFlag MyFlag
}

-- FRAME: Frame delimiter
MyFlag ::= My8BitArray -- Fixed value 0x7E

-- FRAME: Source of frame and destination
MyAddr ::= SEQUENCE {
  destinationAddress MyDestinationAddress,
  sourceAddress MySourceAddress
}

-- ADDR: Destination address
MyDestinationAddress ::= SEQUENCE (SIZE (7)) OF My8BitArray

-- ADDR: Source address
MySourceAddress ::= SEQUENCE (SIZE (7)) OF My8BitArray

-- FRAME: Type of frame
MyControl ::= My8BitArray

-- FRAME: Protocol of info
MyPid ::= My8BitArray -- Fixed value 0xF0

-- FRAME: Info structure
MyInfo ::= SEQUENCE {
  ba MyBa,
  cntrl MyCntrl,
  code MyCode,
  operationData MyOperationData,
  ifcs MyIfcs,
  auth MyAuth OPTIONAL
}

-- INFO: Bus address
MyBa ::= SEQUENCE {
```

```
  srcAddr MySrcAddr,
  dstAddr MyDstAddr
}

-- BA: Source address
MySrcAddr ::= My4BitArray

-- BA: Destination address
MyDstAddr ::= My4BitArray

-- INFO: Control - frame type
MyCntrl ::= My8BitArray -- Fixed value 0x03

-- INFO: Code - operation
MyCode ::= My8BitArray

-- INFO: Operation data
MyOperationData ::= SEQUENCE (SIZE (0..250)) OF My8BitArray

-- INFO: Info Frame check sequence
MyIfcs ::= My16BitArray

-- INFO: Authentication
MyAuth ::= My16BitArray

-- FRAME: Frame check sequence
MyFcs ::= My16BitArray

-- Stack of L3 frames
MyL3Stack ::= SEQUENCE (SIZE (0..255)) OF MyL3Data

-- Stack of L3 frame numbers
MyL3StackNumbers ::= SEQUENCE (SIZE (0..255)) OF BOOLEAN

-- Integer for L3 frame stack numbering
MyL3StackInteger ::= INTEGER (0..254)

-- Stack of L3 missing frame numbers in octets
MyL3MissingStack ::= SEQUENCE (SIZE (0..255)) OF My8BitArray

-- Stack for L3 data in bits
MyL3DataStack ::= SEQUENCE (SIZE (0..1960)) OF BOOLEAN

-- Integer for L3 data stack value numbering
MyL3DataStackValueInteger ::= INTEGER (0..1960)

-- Stack for data of L3 FCS calculation
MyL3FcsStack ::= SEQUENCE (SIZE (80..2040)) OF BOOLEAN  -- L3 frame is empty
or full

-- Integer for data of L3 FCS calculation
```

```
MyL3FcsStackInteger ::= INTEGER (0..2023)

-- Polynomial for L3 FCS calculation
MyL3FcsPolynomial ::= SEQUENCE (SIZE (17)) OF BOOLEAN

-- L3 frame structure
MyL3Frame ::= SEQUENCE {
  l3Ba MyL3Ba,
  l3Cntrl MyL3Cntrl,
  l3Code MyL3Code,
  l3Tte MyL3Tte,
  l3Mrt MyL3Mrt,
  l3Fnum MyL3Fnum,
  l3Data MyL3Data,
  l3Fcs MyL3Fcs
}

-- L3 FRAME: Bus address
MyL3Ba ::= SEQUENCE {
  l3SrcAddr MyL3SrcAddr,
  l3DstAddr MyL3DstAddr
}

-- L3BA: Source address
MyL3SrcAddr ::= My4BitArray

-- L3BA: Destination address
MyL3DstAddr ::= My4BitArray

-- L3 FRAME: Control - frame type
MyL3Cntrl ::= My8BitArray -- Fixed value 0x03

-- L3 FRAME: Code - data frame identificator
MyL3Code ::= My8BitArray

-- L3 FRAME: Time to end
MyL3Tte ::= My16BitArray

-- L3 FRAME: Maximum response time
MyL3Mrt ::= My16BitArray

-- L3 FRAME: Frame counter
MyL3Fnum ::= My8BitArray

-- L3 FRAME: Data
MyL3Data ::= SEQUENCE (SIZE (0..245)) OF My8BitArray

-- Integer for L3 data
MyL3DataInteger ::= INTEGER (0..244)

-- Stack for L3 acknowledgement data in bits
```

```
MyL3AckDataStack ::= SEQUENCE (SIZE (0..1952)) OF BOOLEAN

-- Integer for L3 acknowledgement data stack value numbering
MyL3AckDataStackValueInteger ::= INTEGER (0..1952)

-- Stack for data of L3 acknowledgement FCS calculation
MyL3AckFcsStack ::= SEQUENCE (SIZE (88..2040)) OF BOOLEAN  -- L3
acknowledgement frame is empty or full

-- Stack of L3 missing frame numbers
MyL3AckMissingStack ::= SEQUENCE (SIZE (0..244)) OF INTEGER (0..255)

-- Integer for stack of L3 missing frame stack numbering
MyL3AckMissingStackInteger ::= INTEGER (0..243)

-- L3 FRAME: Frame check sequence
MyL3Fcs ::= My16BitArray

-- L3 acknowledgement frame structure
MyL3AckFrame ::= SEQUENCE {
  l3Ba MyL3Ba,
  l3Cntrl MyL3Cntrl,
  l3Code MyL3Code,
  l3Tte MyL3Tte,
  l3Mrt MyL3Mrt,
  l3Lfn MyL3Lfn,
  l3Hfn MyL3Hfn,
  l3Rrq MyL3Rrq,
  l3Fcs MyL3Fcs
}

-- L3 ACKNOWLEDGEMENT FRAME: Lowest sequential frame number
MyL3Lfn ::= My8BitArray

-- L3 ACKNOWLEDGEMENT FRAME: Highest frame sequence number
MyL3Hfn ::= My8BitArray

-- L3 ACKNOWLEDGEMENT FRAME: Frame numbers missing
MyL3Rrq ::= SEQUENCE (SIZE (0..244)) OF My8BitArray

-- Integer for L3 frame numbers missing
MyL3RrqInteger ::= INTEGER (0..243)

-- Input data structure
MyInputData ::= SEQUENCE {
  ba MyBa,
  code MyCode,
  rawData MyRawData OPTIONAL,
  l3Mrt MyL3Mrt OPTIONAL,
  l3AckMissingStack MyL3AckMissingStack OPTIONAL
}
```

```
-- INPUT DATA: Raw data
MyRawData ::= SEQUENCE (SIZE (0..245)) OF My8BitArray -- Up to 1 frame(s) of
data

-- Integer for raw data numbering
MyRawDataInteger ::= INTEGER (0..245)

-- Acknowledgement data structure
MyAckData ::= SEQUENCE {
  ba MyBa,
  code MyCode,
  l3Mrt MyL3Mrt,
  l3Lfn MyL3Lfn,
  l3Hfn MyL3Hfn,
  l3MissingData MyL3MissingStack
}

END
```