Lehrstuhl für Raumfahrttechnik
Prof. Dr. rer. nat.
Ulrich Walter

Diplomarbeit
RT-DA-2011/20

Integration of the Formation Flying Testbed with the European Proximity Operations Simulator

Author: Florian Rems

Betreuer: Dipl.-Ing. Markus Pietras
Lehrstuhl für Raumfahrttechnik / Institute of Aeronautics
Technische Universität München

Dr.-Ing. Toralf Boge
Deutsches Raumfahrtkontrollzentrum (GSOC)
Einrichtung Raumflugbetrieb und Astronautentraining
Deutsches Zentrum für Luft- und Raumfahrt (DLR)

Bestätigung der eigenständigen Arbeit

Ich erkläre hiermit, dass ich diese Arbeit ohne fremde Hilfe angefertigt und nur die in dem Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

Garching, den _____

Name:	Florian Rems
Matrikelnummer	2823162

Zusammenfassung

Der European Proximity Operations Simulator EPOS (Teil des GSOC, Oberpfaffenhofen bei München) ermöglicht die Simulation von Rendezvous und Docking (RvD) Szenarien mit zwei Raumfahrzeugen und entsprechender Sensor Hardware, die "Hardware-in-the-Loop" in das System eingschleift werden kann. Mockups der Raumfahrzeuge werden auf zwei Industrieroboter montiert. Beide Raumfahrzeuge können in sechs Freiheitsgrden bewegt werden, wodurch die relative Orientierung und Position simuliert wird. Ein Echtzeit-Kontrollsystem, das das Echtzeit Betriebssystem VxWorks in Kombination mit dem Real-Time Workshop von Matlab/Simulink nutzt, kommandiert die Roboter. Diese Simulationsumgebung erfordert es, dass jeder Kunde, will er EPOS nutzen, seine Simulationscode oder gar sein ganzes Simulationssystem (Hardware und Software) an EPOS Anforderungen anpassen muss. Im Detail bedeutet dies einen erheblichen Aufwand an Zeit und Ressourcen, der den Vorteil einer Hardware in the Loop Simulation mit EPOS zunicht machen würde.

Es ist das Ziel dieser Arbeit, diesen Aufwand deutlich zu reduzieren. Dafür soll ein Software-Packet entwickelt, implementiert und getestet werden, das es erlaubt, einen externen Satellitensimulator über Ethernet mit dem EPOS Echtzeitkontrollsystem zu verbinden. Obgleich diese Strategie viele Probleme löst, bringt sie auch neue hervor. Ein Beispiel dafür sei die Tatsache, dass ein nicht-deterministisches Netzwerk, wie Ethernet eines darstellt, in Verbindung mit einer Echtzeitumgebung genutzt wird.

Um diesen Problemen beizukommen, wird ein Kommunikationsprotokoll (der Netzwerk-Applikationsschicht) entwickelt, das speziell auf die Anforderungen von EPOS ausgerichtet ist. Es besteht aus zwei Sub-Protokollen, dem Simulation Connection Protocol (SCP) und dem Remote Simulation Protocol (RSP). Neben vielen anderen Aufgaben, realisieren diese Protokolle eine Datenverbindung zwischen zwei Simulink Modellen, überwachen die Latenz der Datenpakete, regeln die Zusammenarbeit zwischen externem Simulator und EPOS (Herstellen der Anfangsbedingungen, Timing...) und interpolieren die Robotertrajektorien zwischen den Kommandos des externen Simulators. SCP/RSP wird sodann in Form von Simulink S-Functions implementiert. Dabei sind die Simulink Blöcke nicht nur mit Windows kompatibel, sondern auch mit VxWorks (Simulink Real-Time Workshop).

Schließlich wird das System getestet, wobei ein Demonstrationsszenario benutzt wird, das auf einem Formationsflug-Testbett ausgeführt wird. Diese Simulationsumgebung ist ein Multi-Satelliten Simulator, der von der Formationsflug-Gruppe am GSOC enwickelt wurde. Es simuliert Formationsflug-Algorithmen mit der Möglichkeit, echte GPS-Empfänger in Kombination mit einem GPS Signalsimulator sowie einen auf der SPARC Architektur basierenden Echtzeit-Computer als On-Board-Computer einzuschleifen. Die Auswertung dieses Demoszenarios beweist, dass die Software, die in dieser Arbeit entwickelt und implementiert wurde, in der Praxis tatsächlich funktioniert und die gestellten Anforderungen erfüllt. Es wird gezeigt, dass die Verbindungsqualität im lokalen EPOS Netzwerk ausreicht, um Simulationen mit externen Simulatoren via Ethernet durchzuführen, vorausgesetzt die Sample-Frequenz des externen Simulators ist nicht zu groß. Weiterhin zeigt der Drift, d.h. das Zeit-Differential zwischen den Uhren des externen Simulators und der von EPOS, eine Größenordnung, die der Genauigkeit der Timer-Hardware sehr nahe kommt. Daher kann eine Simulation, bestehend aus einem externen Simulator und EPOS, problemlos über mehrere Stunden durchgeführt werden.

Die Software trägt zur Flexibilität EPOS' bei. Zuvor war die Realisierung einer von 0 verschiedenen Startgeschwindigkeit und -drehrate nur sehr umständlich und aufwendig zu realisieren. SCP/RSP löst diese Aufgabe automatisch, indem eine individuelle Starttrajektorie berechnet wird, wann immer dies notwendig ist. Software, die auf dem EPOS Echtzeitkontrollsystem läuft, muss mit einer Sample-Frequenz von $250Hz$ ausgeführt werden. SCP/RSP erlaubt es, durch translatorische und rotatorische Interpolation einen externen Simulator mit einer wesentlich niedrigeren Sample-Frequenz zu betreiben. Und schließlich vereinfacht SCP/RSP den Simulationsprozess. Verschiedene externe Simulationen können gestartet und gestoppt werden, ohne dass die EPOS Echtzeitsimulation neu gestart werden muss.

Abstract

The European Proximity Operations Simulator EPOS (part of GSOC, Oberpfaffenhofen near Munich) allows to simulate Rendezvous and Docking scenarios involving two spacecraft and integrating rendezvous sensors (Hardware-in-the-Loop). Mockups of the spacecraft are mounted to two industrial robots. Both spacecraft can be moved in six degrees of freedom each, thus simulating relative orientation. A real-time control system operates the robots, involving the Real-Time Operating System VxWorks combined with Matlab/Simulink Real-Time Workshop. This environment demands that any customer adapt his simulation code, satellite simulator etc. to EPOS' needs. In detail, this involves considerable effort and time which may obliterate the benefits of a Hardware-in-the-Loop simulation with EPOS.

It is the aim of this thesis to reduce this effort distinctly by designing, implementing and testing a software package which connects any external satellite simulator via ethernet to the EPOS real-time control system. Although this stategy solves many problems, new ones are created, like the fact that a non-deterministic network, like ethernet, is used in a real-time environment.

To cope with these problems, an application layer communication protocol is developed, specifically tailored to meet EPOS' needs. It comprises two subprotocols, the Simulation Connection Protocol (SCP) and the Remote Simulation Protocol (RSP). Among many other tasks, these protocols realize a data connection between two Simulink models, monitor packet delay, manage the interaction between a remote simulator and EPOS (realization of starting conditions, timing...) and interpolate the robot trajectory in-between remote simulator commands. These communication protocols are then implemented in the form of Simulink S-Functions, not only compatible to Windows but also to the Real-Time Operating System VxWorks.

The remote control system is tested using a demo scenario running on a Formation-Flying-Testbed. This simulation environment is a multi-satellite simulator developed by the Formation-Flying group at GSOC. It runs Formation-Flying algorithms with the possibility to use real GPS receivers (in combination with a GPS signal simulator) and a SPARC based real-time computer to serve as On-Board Computer. Evaluation of this demo-scenario is presented. It proofs, that the remote control system designed and implemented in this thesis indeed works and that it fullfills the requirements. It is illustrated, that connection quality in the local EPOS network allows to couple a remote simulation with EPOS via ethernet, as long as the remote simulator's sample frequency is not too large. Moreover, drift, e.g. the time differential between the remote simulator's clock and the EPOS real-time clock, shows to be in the limits of timer hardware precision. As a result, remote simulations can be run for many hours before drift becomes a problem.

The remote simulation software adds to the flexibility of EPOS. Before, an initial speed and angular velocity different from zero was inconvenient to realize. SCP/RSP carries out this task automatically, by determining an initial trajectory when needed. Software running on the EPOS real-time control system has to run at a frequency of $250Hz$ obligatorily. SCP/RSP allows to run a remote simulation at a much lower frequency by translational and rotational interpolation. And SCP/RSP simplifies the simulation process. Different remote simulations can be started and stopped without the need to restart the EPOS real-time simulation.

Contents

List of Figures

List of Tables

Symbols

$\boldsymbol{a}^{SECI \to CLW}$	–	rotation axis associated with coordinate transformation
$Body$	–	body-fixed frame
c	–	subscript associated with RemoteSim-Client or chaser S/C
C	–	RemoteSim-Client time frame
$\boldsymbol{F}$	–	force
H	–	SimCon home instance time frame
k_n	–	general trajectory value at timestep n
l	–	number of elements in FIFO
M	–	rotation matrix
n	–	simulation step
n_{acqu1}	–	simulation step of forward packet data's acquisition
n_{curr}	–	current time step
n_{hit}	–	next hit
n_{hit+1}	–	next-but-one hit
$n_{init,c}$	–	initial time step of RemoteSim-Client
$n_{init,s}$	–	initial time step of RemoteSim-Server
n_{offset}	–	simulation step offset
n_{out}	–	simulation step of packet data's output
n_{recv}	–	time step when a packet is received
n_{ref}^{C}	–	RemoteSim-Client reference time step
n_{ref}^{H}	–	home simulation reference time step
n_{ref}^{S}	–	RemoteSim-Server reference time step
n_{ref}^{T}	–	target simulation reference time step
Δn_{dead}	–	DeadTime in simulation steps

q	–	rotation/attitude quaternion
q_c	–	attitude quaternion of chaser spacecraft
q_{curr}	–	current attitude
q_{req}	–	requested attitude
q_t	–	attitude quaternion of target spacecraft
$\boldsymbol{r}$	m	position vector
r	m	position vector quaternion
$\boldsymbol{r}_c$	m	position vector of chaser spacecraft
r_c	m	position vector quaternion of chaser spacecraft
$\boldsymbol{r}_{curr}$	m	current position
$\boldsymbol{r}_{req}$	m	requested position
$\dot{\boldsymbol{r}}_{curr}$	m/s	current speed
$\dot{\boldsymbol{r}}_{req}$	m/s	requested speed
$\boldsymbol{r}_t$	m	position vector of target spacecraft
r_t	m	position vector quaternion of target spacecraft
S	–	RemoteSim-Server reference time step
t	–	subscript associated with target S/C
t_{acqu0}	s	time of data acquisition for return packet
t_{acqu1}	s	time of data acquisition for forward packet
t^S_{end}	–	end time of interpolation at RSP server
t^H	s	some point in time in home computer time frame
t^H_j	s	timestep j of simulation on home computer
t_{out}	s	time of a packet data's output
t^H_{ref}	s	reference time on home computer
t_{recv0}	s	time of return packet's reception
t_{recv1}	s	time of forward packet's reception
t^S_{start}	–	start time of interpolation at RSP server
t^T	s	some point in time in target computer time frame

t_i^T	s	timestep i of simulation on target computer
t_{ref}^T	s	reference time on target computer
$t^{T \to H}$	s	time transformation from target to home time frame
t_{trm0}	s	time of return packet's transmission
t_{trm1}	s	time of forward packet's transmission
Δt_{a2o}	s	Acqu2Output
Δt_{dead}	s	DeadTime in seconds
Δt_{init}	s	timespan for initial trajectory
Δt_{interp}	s	timespan between start and end point of interpolation
Δt_{ping}	ms	timeout for determining a broken SCP link
Δt_{s2r}	s	Send2Receive
Δt_{sample}	s	sampletime
T	–	SimCon target instance time frame
$\boldsymbol{T}$	Nm	torque
U	–	user frame, either ECI or CLW
$\hat{\boldsymbol{X}}_{CLW}^{ECI}$	–	CLW x-axis unit vector
$\hat{\boldsymbol{Y}}_{CLW}^{ECI}$	–	CLW y-axis unit vector
$\hat{\boldsymbol{Z}}_{CLW}^{ECI}$	–	CLW z-axis unit vector
ν	–	forerun
$\varphi^{SECI \to CLW}$	–	rotation angle associated with coordinate transformation
$\boldsymbol{\omega}_{curr}$	rad/s	requested angular velocity
$\boldsymbol{\omega}_{req}$	rad/s	requested angular velocity

Abbreviations

6DOF	6 Degree Of Freedom
ACS	Application Control System
API	Application Programming Interface
BCS	Base Coordinate System
BD	Backward Difference
CD	Central Difference
CLW	Clohessy-Wiltshire Coordinate System
CDGPS	Carrier Phase Differential GPS
ECI	Earth-Center-Inertial Coordinate System
SECI	Shifted-Earth-Center-Inertial Coordinate System
EPOS	European Proximity Operations Simulator
FCC	Flight Control Computer
FD	Forward Difference
FF	Formation Flying
FIFO	First-In-First-Out
FMC	Facility Monitoring and Control
FTP	File Transfer Protocol
GLB	Global Lab Coordinates
GNC	Guidance Navigation and Control
GPS	Global Positioning System
GSOC	German Space Operations Center
GSS	GNSS Signal Simulator
HiL	Hardware in the Loop
HTTP	Hypertext Transfer Protocol
IDC	Ideal Coordinate System

IJT	Ideal Joint Coordinates
I/O	Input/Output
IP	Internet Protocol
KRC	Kuka Robot Control
LAN	Local Area Network
LRC	Local Robot Control
MCU	Mobile Control Unit
MMI	Man Machine Interface
OBC	Onboard Computer
POV	Point Of View
RF	Radio Frequency
RSI	Robot Sensor Interface
RSP	Remote Simulation Protocol
RT	Real Time
RTOS	Real-Time Operating System
RTSP	Real-Time Streaming Protocol
RTW	Real-Time Workshop (Matlab/Simulink)
RvD	Rendezvous and Docking
RTW	Real Time Workshop
S/C	spacecraft
SCP	Simulation Connection Protocol
SNR	Signal to Noise Ratio
SPARC	Scalable Processor Architecture
TCP	Transmission Control Protocol
TCS	Tool Coordinate System
TLC	Target Language Compiler
UDP	User Datagram Protocol

1. Introduction

1.1. Overview

The European Proximity Operations Simulator (EPOS) is part of the German Space Operations Center (GSOC), an institute of the German Aerospace Center (DLR), located in Oberpfaffenhofen, near Munich. EPOS is a sophisticated facility for simulating Rendezvous and Docking (RvD) scenarios between two spacecraft. For that purpose, spacecraft mockups can be mounted to two KUKA robots with 6 degrees of freedom each. With one of the robots mounted on a linear slide, complete relative motion of the spacecraft can be realized. The facility is controlled with a set of real-time computers, running the Real-Time Operating System VxWorks in combination with Matlab/Simulink Real-Time Workshop. Unusal for industrial robot applications, trajectories are commanded with a high frequency of $250Hz$. EPOS is a Hardware-in-the-Loop (HiL) system. Trajectories can be generated in real-time in the form of a Simulink model. Thus, it is possible to use the data from some kind of sensor (i.e. a camera), process the information and have the robots move accordingly, i.e. one spacecraft shall keep a specific relative position and orientation. EPOS' high precision and numerous possibilities necessitate an accordingly complex and sophisticated control system. Such a control system enlarges the effort for external customers to adapt their code and simulation evironments to EPOS' needs. Among many problems, compatibilty to the RTOS VxWorks and capability of running at a sample frequency of 250Hz (most satellite flight software runs at a much lower frequency, including mockup On-Orbit Computers) are only two of them. This gives rise to the idea of *connecting* any external simulator which has been developed by some customer to the EPOS real-time control system, rather than disassembling, modifying and *integrating* it into the control system. Since ethernet is an established standard and already available in the form of the EPOS local network, it suggests itself as the interface to be used. In that way, the customer's simulation model does not have to be ported to VxWorks and it can be executed with a much lower frequency. Many other problems are solved this way. But others arise. Ethernet is non-deterministic which counteracts real-time operability. If the external simulator commands the robots with a lower sample frequency, robot trajectories have to be interpolated. Another question is how to realize robot starting conditions involving initial speed and acceleration. It is the aim of this thesis to solve all these problems, thereby realizing a remote control connection between an external satellite simulator and the EPOS real-time operated facility. This comprises definition of remote control requirements, design of communication and interaction procedures including tasks like interpolation and coordinate transformation, implementation of the system in the form of Simulink S-Functions considering the special attributes of a RTOS like VxWorks and simulation of a demo scenario to illustrate proper functionality of the developed software. For the last task, the Formation-Flying-Testbed (FF-Testbed) shall be used. This is a multi satellite simulator developed by the Formation-Flying group at GSOC. It consists of two WinXP PCs, a GPS signal simulator and Pheonix GPS receivers. Also, a mockup On-Board Computer can be included as real flight hardware. The demo scenario realized as a Simulink model running on the FF-Testbed is provided by the Formation-Flying group.

In this thesis, two communication protocols are designed, constituting an application layer protocol in terms of computer network communication. The Simulation Connection Protocol (SCP) realizes a general connection between two Simulink models running on different hosts. Among other tasks, SCP also monitors connection quality by evaluating the delay it takes for data packets to be transmitted from one host to another. The Remote Simulation Protocol (RSP) manages the interaction between an external satellite simulator and EPOS. Therefore, the protocol distinguishes a RSP client (external simulator) and a RSP server (EPOS real-time simulation). It deals with the timing between both end-points, thus realizing starting conditions and seamless transition to the simulation phase. It interpolates robot trajectory C^1 continuously (position and attitude) to avoid peaks in robot acceleration. Furthermore, it deals with broken connections, thereby slowing the robots down safely without stopping the EPOS real-time simulation.

The protocols are implemented as Simulink c-mex S-Functions. One block, the RemoteSim-Client, comprises SCP and the client part of RSP. The other block, RemoteSim-Server, comprises SCP and the server part of RSP. Implementation is carried out such that most part of the software could be used independently from Simulink in another project and on another platform. Simulink/S-Function specific code constitutes only the minor portion. The programming language is C/C++. While RemoteSim-Client is implemented for Windows only, RemoteSim-Server runs on Windows and the RTOS VxWorks. Especially implementation for VxWorks showed to demand great effort, since numerous special attributes of Real-Time Operating Systems have to be considered.

Subquently, a brief overview of this thesis' chapters is given.

- **Chapter 1 - Introduction**: The EPOS facility is described in detail. This comprises the hardware setup, e.g. the robots and other hardware elements, a presentation of different coordinate systems used and a thorough description of the EPOS real-time control system. Moreover, facility operation procedures are outlined. These facts and relations are of paramount importants for the subsequent chapters. Another section gives an overview of Formation-Flying concepts and presents the FF-Testbed. Finally, some notes about symbol notation used in this thesis are made.
- **Chapter 2 - Remote Control Architecture**: This chapter begins with some compressed information about computer networks. This is followed by a presentation of EPOS remote control requirements, wherin all main problematic areas are covered and of which only few have been mentioned above. A top level overview of the two communication protocols developed in this thesis is given. Thereafter, these protocols, the Simulation Connection Protocol (SCP) and the Remote Simulation Protocol (RSP) are presented in detail. Protocol parameters, network packet structures and behaviour is described in detail. Within the context of RSP, the interaction between EPOS and a remote simulation is outlined.
- **Chapter 3 - Remote Control Implementation**: A top-level outline of the ample software which implements remote control architecture is given. The responsibilities of the different C++ classes involved is outlined. The chapter is not intended to serve as complete software documentation but to make clear the overall structure and working principle of the implementation. A detailed description is given in appendix C.
- **Chapter 4 - Remote Control Analysis**: By evaluating a Formation-Flying demo scenario, it is illustrated that RemoteSim indeed works and that the requirements are fullfilled. Accurate behaviour of RSP, seamless transition from initial trajectory to simulation and correct trajectory interpolation is shown. Moreover, connection quality and drift are examined.
- **Chapter 5 - Conclusion**: The chapter briefly summarizes the results of EPOS remote control developement. Moreover, possible enhancements and future extensions of the remote control software are given.
- **Appendix**: In appendix A and B, inputs, outputs and parameters of the created Simulink blocks are listed with information about its relation with SCP/RSP as well as data format. Appendix C serves as documentation of the developed software. It comprises an overview of C++ class structure as well as a detailed descriptions, outlines of context and UML diagrams of all classes. Its purpose is not only to document the software as a complete system, but to provide the necessary information to reuse individual classes in other projects. Appendix D gives practical tipps on building RemoteSim for Simulink, on modelling a RemoteSim-Client and RemoteSim-Server simulation and on the general simulation procedure. Appendix E lists all header and source files with its contents, as well as other files as part of the software.

1.2. The European Proximity Operations Simulator (EPOS)

The European Proximity Operations Simulator is a Hardware-in-the-Loop (HiL) simulator for the last phase (25m to 0m) of Rendezvous and Docking (RvD) scenarios involving two spacecraft. While physical simulation of satellite dynamics on ground is virtually impossible, its accurate numerical simulation is much easier and leads to very good results. And while a realistic sensor output (camera, sensor for contact forces...) is hard to simulate numerically, real sensor hardware can easily be used on ground. EPOS profits from combining numerical models of satellite dynamics and real hardware for RvD simulations.

Figure 1.1.: EPOS Facility ([5])

This section is intended to give a detailed overview of EPOS. Knowledge of structure and function of the simulation facility are of vital importance for this diploma thesis. The description presented in this section is based on [1],[2],[3],[4],[5] and [6].

1.2.1. Hardware Setup

The hardware setup of EPOS is depicted in Fig. 1.2. The central elements are two standard 6DOF

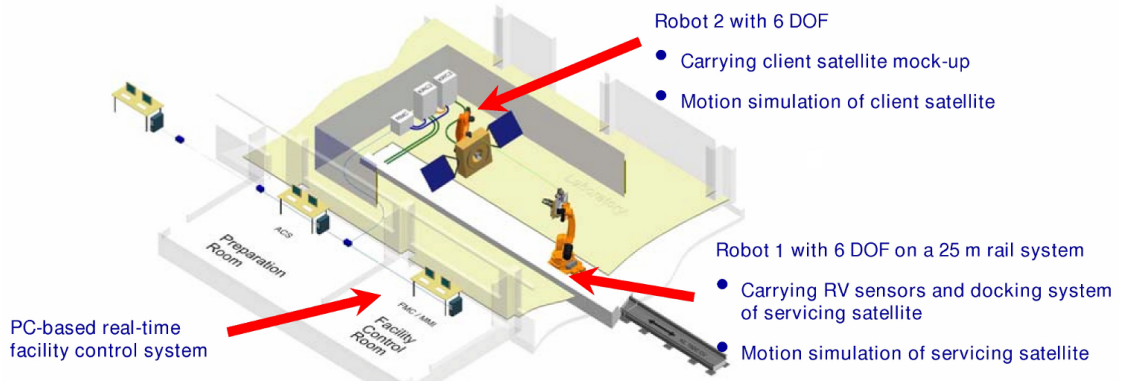

Figure 1.2.: Hardware Setup of EPOS Facility ([5],modified)

industrial robots. One of them, a KUKA KR100HA (robot 2 in Fig. 1.2), is mounted on a linear rail, which is 25 m long. The whole robot can be moved on that rail in order to simulate the approach of two satellites in space. Thus the linear rail consitutes another degree of freedom. The other robot is a KUKA KR240 and is mounted at the end of the rail, its base fixed in the laboratory. Each robot is equipped with a breadboard attached to the tool flange which can be used to mount satellite mockups or sensor devices. Furthermore power supplys with different voltages as well as various data interfaces are available (see [5] for a specifications of the interfaces).

In a typical scenario, as shown in Fig. 1.2, the mockup of the client satellite is mounted on the front panel of robot 1. The servicing satellite is then represented by robot 2, which may carry sensor equippement and docking systems on its front panel. Angular and relative transversal movement of

both satellites can then be simulated by moving the 6DOF robots and including the linear rail as a 13th DOF.

1.2.2. EPOS Coordinate Systems

A number of different coordinate systems are used to operate EPOS.

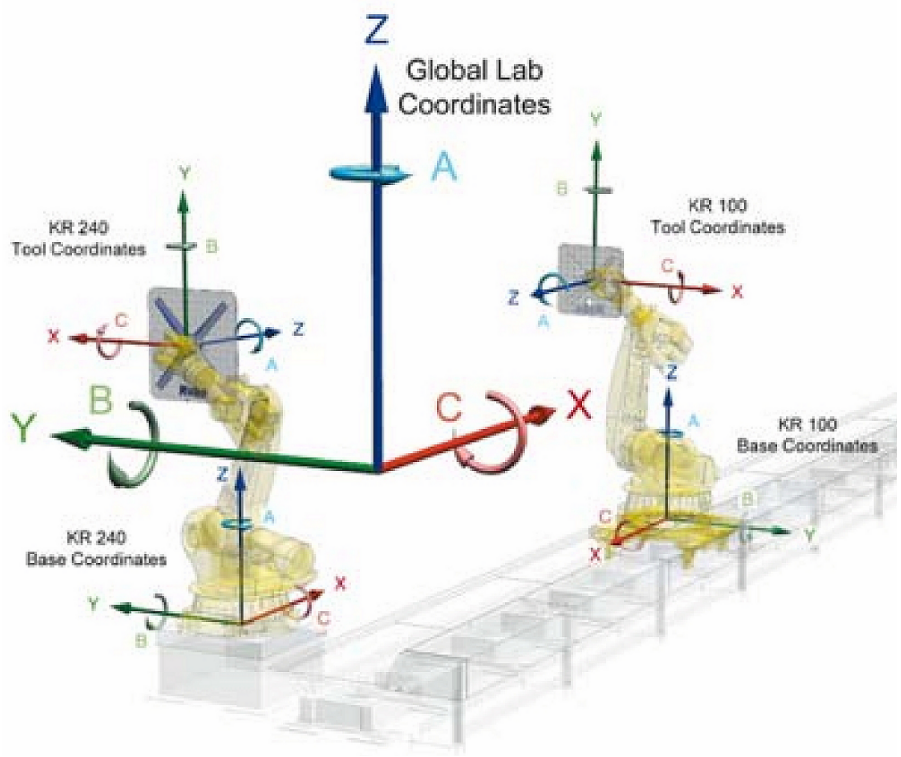

Figure 1.3.: EPOS Coordinate Systems ([5])

- **Axis Coordinates**: Each robot axis has its own coordinate system represented by the axis angle. Commanding angles directly can be useful for operating the robots manually, e.g. if sensors are to be mounted. Axis Coordinates are equal to Ideal Joint Coordinates (IJT).
- **Tool Coordinate System (TCS)**: As illustrated in Fig.1.3, the TCS is a cartesian coordinate system fixed to the robot flange. If a satellite mockup is mounted, the TCS can be considered to be somewhat similar to a body-fixed-frame of the satellite. The TCS' origin lies in the middle of the tool flange. The Z-axis is perpendicular to the breadboard's mounting face. X-axis and Y-axis are oriented as depicted to form a right-handed-side coordinate system.
- **Base Coordinate System (BCS)**: The cartesian BCS is fixed to the base of the robot, see Fig.1.3. Its origin lies in the middle of the base. The Z-axis points uppwards and the X-axis is parallel to the linear slide, directed towards the other robot. The Y-axis completes the right-handed-side coordinate system. The BCS is equal to Ideal Device Coordinates (IDC).
- **Global Lab Coordinates (GLB)**: As the name suggests, these cartesian coordinates are fixed with respect to the laboratory. Its orientation is similar to that of the chaser robot (the fixed robot) with a well-defined origin located in the laboratory. It may be required to position a robot flange with respect to the laboratory. In such a case, GLB is useful as reference coordinate system.
- **Clohessy-Wiltshire Coordinates (CLW)**: This well-known coordinate system is used to command satellite position relative to some reference frame, which may be the body-fixed-frame of the client satellite or some "'orbiting"' reference frame. Position and attitude of this so-called

Point Of View (POV) coordinate system with respect to GLC have to be commanded along with the spacecrafts' positions and attitudes. Thus, the user can place the coordinate system he chose in the laboratory as he pleases.

1.2.3. Facility Control

Control of the EPOS facility and simulation of RvD scenarios is achieved using a set of computers, each fulfilling a specific task in the sytem and being connected via different types of networks. Facility control can be structured by associating its components with different layers or levels. Following this philosophy, Fig.1.4 gives an overview of facility control.

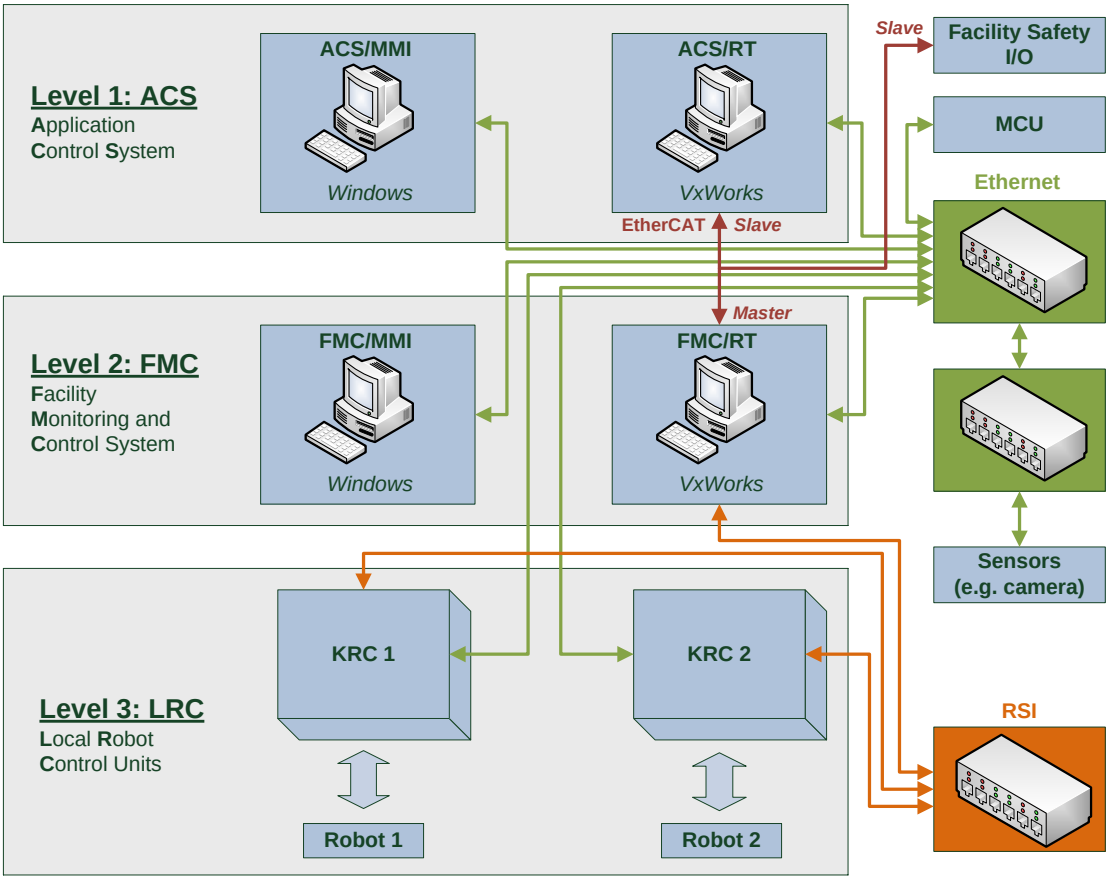

Figure 1.4.: EPOS Facility Control

Three levels can be identified. The first level corresponds to the Application Control System (ACS). The second Level is associated with the Facility Monitoring and Control System (FMC). And finally the third level encompasses the Local Robot Control Units (LRC). Both the ACS and the FMC are comprised of two PCs, a Man Machine Interface (MMI) running Windows XP and a Real Time computer (RT) running Windriver VxWorks, respectively. The LRC consists of two identical KUKA Robot Controllers (KRC) which are directly connected to the particular robots. Somewhat outside of this level representation there are two Mobile Control Units (MCU), the Facility Safety I/O and sensors which can be used for HiL simulations if mounted to the robots' breadboards.

All these components (except the Facility Safety I/O) are connected by an ethernet Windows network via switch. To fullfill realtime requirements ACS/RT and FMC/RT are connected by EtherCAT, where the FMC/RT serves as master and the ACS/RT as slave. Another slave in this EtherCAT network is the Facility Safety I/O. For connecting both KRCs with the FMC/RT another network is used, the so called Robot Sensor Interface (RSI) from KUKA.

There are two different modes to command EPOS. Synchronous and Asynchronous. In order to understand how the components shown in Fig.1.4 interact, let's look at the data flow during these operational modes.

Asynchronous Mode: This mode is used to have the robot follow a predefined trajectory which is given by a file with specified format stored at the FMC/MMI Windows computer. The FMC/MMI provides the data in the file to its realtime counterpart, the FMC/RT, via ethernet. Suppose robot positions are commanded using a CLW frame. In such a case the FMC/RT first transforms these coordinates to Global Lab Coordinates and then to Base Coordinates. Subsequently the BCS values sent to the KUKA Robot Controllers. There they are transformed to Axis Coordinates (Ideal Device Coordinates) which are finally given to the robots. The KRCs also provide current robot positions back to the FMC/RT, which are transformed in opposite order and thus provided to the user in the specified coordinates.

Synchronous Mode: In contrast to the asynchronous mode the trajectory is not defined beforehand. Rather, robot positions are calculated at runtime by the RvD simulation. Therefore the ACS/MMI and ACS/RT computers are needed. On the Windows computer (ACS/MMI) Simulink Real-Time-Workshop for VxWorks can be used to create a RvD simulation. Interface Simulink blocks allow to integrate robots and sensors into the simulation to realize a HiL scenario. Simulink Real-Time-Workshop builds the realtime code from the Simulink model for VxWorks. This code is then loaded to the VxWorks realtime computer (ACS/RT) via ethernet. While the simulation is running, ACS/RT sends position and attitude commands to the FMC/RT in realtime using the EtherCAT interface. The FMC/RT then processes the data according to asynchronous mode, as described above.

The robots can be commanded not only using CLW coordinates, but also GLB, BCS and IJT coordinates. If using these "'lower level"' frames, the FMC/RT just leaves out the according coordinate transformations which are not needed in such a case.

There is also the possibility to move the robots manually with a Mobile Control Unit (MCU). These are handheld devices and can be used in the laboratory hall.

As mentioned above, ACS/MMI and FMC/MMI are Windows computers with the usual desktop interface. However, ACS/RT and FMC/RT are running VxWorks, a Real Time Operating System (RTOS). A graphical user interface is neither available nor necessary. Interaction with the realtime computers is carried out exclusively via the presented networks.

The purpose of the FMC/MMI goes beyond providing trajectory data in asynchronous mode. The software "FMC CMD Center" allows to chose operational mode, switch the robots on and off and control the facility state. Moreover, it realizes a thourough monitoring of EPOS. A logging functionality makes it possible to record robot states during a simulation. Finally, the facility can be operated in simulation mode, that is the robots are not physically involved, but their movement - "as it would be" - is simulated. This allows testing of a Simulink model to detect possible range violations, axes singularities etc.. Another program is the "Security Display Application". All components of the security system (security fence, doors, emergency stops) can be monitored. Finally, a virtual 3D presentation of both robots can be displayed. This is especially useful when the facility is operated in simulation mode.

1.2.4. Synchronous Mode Procedure

The synchronous mode plays the central role in the remote control system developed in this thesis. Therefore it follows a short presentation of the procedure from a user's point of view.

On the ACS/MMI the user has created a Simulink model. This model necessarily contains a block called CMD inteface which represents the connection to both robots. The CMD interface has various inputs and outputs. The one essential in this section is the enable signal. The user builds the realtime code of the simulink model, loads it onto the ACS/RT by clicking *Connect* in Simulink and starts the simulation. At this point, the enable signal is 0. The user clicks *Sync* in the FMC Command Center at the FMC/MMI and the facility changes to state *Move to Start*. In this state, EPOS takes the position signals which are fed to the inputs of the CMD interface block at the time when *Sync* is clicked and moves the robots to these positions and attitudes. As soon as this is achieved, the button *Confirm Sync* is highlighted in the FMC Command Center. A click on this button has the facility finally change to synchronous mode. It is at this point in time that the aforementioned enable signal goes from 0 to 1. This time step marks the actual beginning of the simulation. The Simulink model must consider this and use the enable signal as a starting trigger.

1.2.5. EPOS HiL Principle

This short section is intended to give a brief example of how an HiL simulation is realized with EPOS.

Suppose we have some rendezvous scenario. A chaser satellite has to stay at a specified distance from a target satellite. Also, the chaser satellite is to match its own orientation to the target's attitude. This task shall be achieved by vision based navigation, that is a camera takes pictures of the target satellite. An appropriate image processing algorithm determines the target satellite's orientation and an attitude control algorithm adapts the chaser's attitude.

With EPOS the camera could be mounted to the chaser robot's breadboard. It sends pictures via ethernet to the ACS/RT, which realizes the image processing algorithm, the attitude control algorithm as well as a dynamic simulation of both satellites in orbit. The results of this simulation, position and attitude of both satellite, is then given to the FMC/RT via EtherCAT for the above mentioned coordinate transformations. The RSI interface is used to transfer the position data to the KRCs, which finally move the robots. The effect of this movement is a changed camera picture. Hence the image processing algorithm creates an according output ... Camera and robots are integrated "in the loop" in the simulation.

1.3. Formation Flying and the FF-Testbed

This section gives an overview of the Formation-Flying (FF) concept. Furthermore, the FF-Testbed at DLR is described in detail. The subsequent facts are taken from [8], [9].

1.3.1. Formation Flying

D'Amico gives a striking definition of FF in his PhD thesis [8]: "Formation flying is commonly identified as the collective usage of two or more cooperative spacecraft to exercise the function of a single monolithic virtual instrument." Such a system can thereby go beyond the possibilities of a single satellite. Functions to be performed and associated payloads are distributed over a number of satellites. Larger baselines are achievable and a high degree of redundancy can be realized.

While FF by ground control comes along with strong limitations concerning ground station visibility, autonomous FF allows for higher performance and instant response to contingencies as well as a comparatively high spatial and temporal resolution. However, autonomous FF requires proper on-board computing capabilities.

Typically the following applications for FF applications can be distinguished: Technology demonstration missions (e.g. PRISMA), synthetic aperture interferometers and gravimeters (e.g. TanDEM-X), dual spacecraft telescopes (e.g. XEUS) and multi-spacecraft interferometers (e.g. DARWIN).

To fullfill these task high precision GNC (Guidance Navigation and Control) technologies are used, like for example Carrier Phase Differential GPS (CDGPS).

In order to simulate alike scenarios an earth, a multi satellite simulator based on Matlab/Simulink has been developed by the FF-Team at GSOC. The following section describes this Formation-Flying-Testbed.

1.3.2. FF-Testbed

This section is based on [13]. As Fig.1.5 illustrates, the FF-Testbed is comprised of four main components which are subsequently described briefly.

The Flight Control Computer (FCC) is a Windows XP PC running a Matlab/Simulink model. It simulates spacecraft dynamics, considering in detail among others gravity, drag, relativistic effects and solar radiation pressure. Also, the FCC simulates the behaviour of spacecraft actuators and sensors. Moreover, it allows for the definition of time-dependent events. The Flight Control Computer can handle two S/C independently. Finally, there is an emulation of ground segment functionality available. Thus, telecommands can be sent and telemetry be received.

The GNSS Signal Simulator (GSS) consists of a Spirent GSS7700 device and a PC running SimGEN which steers it. The GSS simulates two independent RF signals as it would be received by a spacecraft's antenna. The signal is based on the S/C's position, speed and attitude provided by the FCC as well

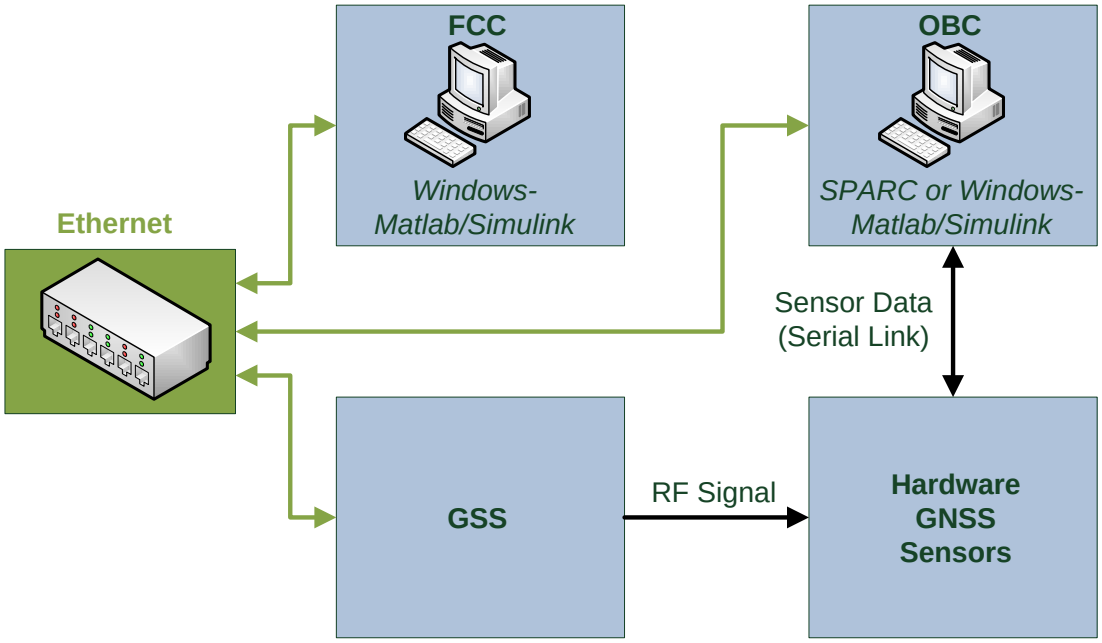

Figure 1.5.: Formation Flying Testbed

as on GPS constellation, atmosphere model etc.. Furthermore the GSS supplies a synchronization signal of $1Hz$ via TCP/IP.

Two single frequency Phoenix GPS receivers (see[14]) constitute the hardware GNSS sensors. They get its RS signals from the GSS and exchange data with the OBC using a serial link.

The Onbard Computer (OBC) runs the GNC flight software. In the configuration present during this thesis, the OBC is realized as a WinXP computer running a Matlab/Simulink model of the flight software. But it is also possible to use a SPARC architecture combined with Matlab/Simulink Real-Time Workshop.

FCC, GSS and OBC are connected via ethernet. The FCC sends telecommand and simulated measurements to the OBC. It also sends motion commands to the GSS, so that the RF signals are consistent with the simulated spacecraft states. The OBC returns telemetry and control actions (for example for attitude control thrusters) to the FCC. The GSS supplies the Phoenix GPS receivers with the RF signals. A serial link between these receivers and the OBC allows for exchange of sensor data.

As mentioned above, the GNSS Signal Simulator outputs a synchronization signal. This signal is used to synchronize FCC and GSS. Moreover, the FCC sends another synchronization trigger using TCP/IP to the OBC. Thus GSS, FCC and OBC can be considered to run in realtime, though the non-deterministic ethernet connection limits this to a certain sample frequency. From [13], a sample frequency above $10Hz$ is not recommended.

1.4. The Idea - Expanding EPOS with an external Simulator via Ethernet

Consider the following scenario: Some research group has available some kind of simulator system specifically designed to support the developement of a space camera, a docking mechanism etc. . This simulator may be a realtime capable system, using an "exotic" RTOS or it may run under soft real-time conditions using WinXP. There may by hardware components (mockups) used in the loop. The simulation may have been designed with a model-based approach, for example a Simulink model, or with a low-level standard programming language like C/C++, Fortran etc.. A perfect example is the FF-Testbed.

The usual way of integrating this simulation with EPOS, and to be more precise with the Simulink model running on the ACS/RT, requires a whole set of problamitc tasks.

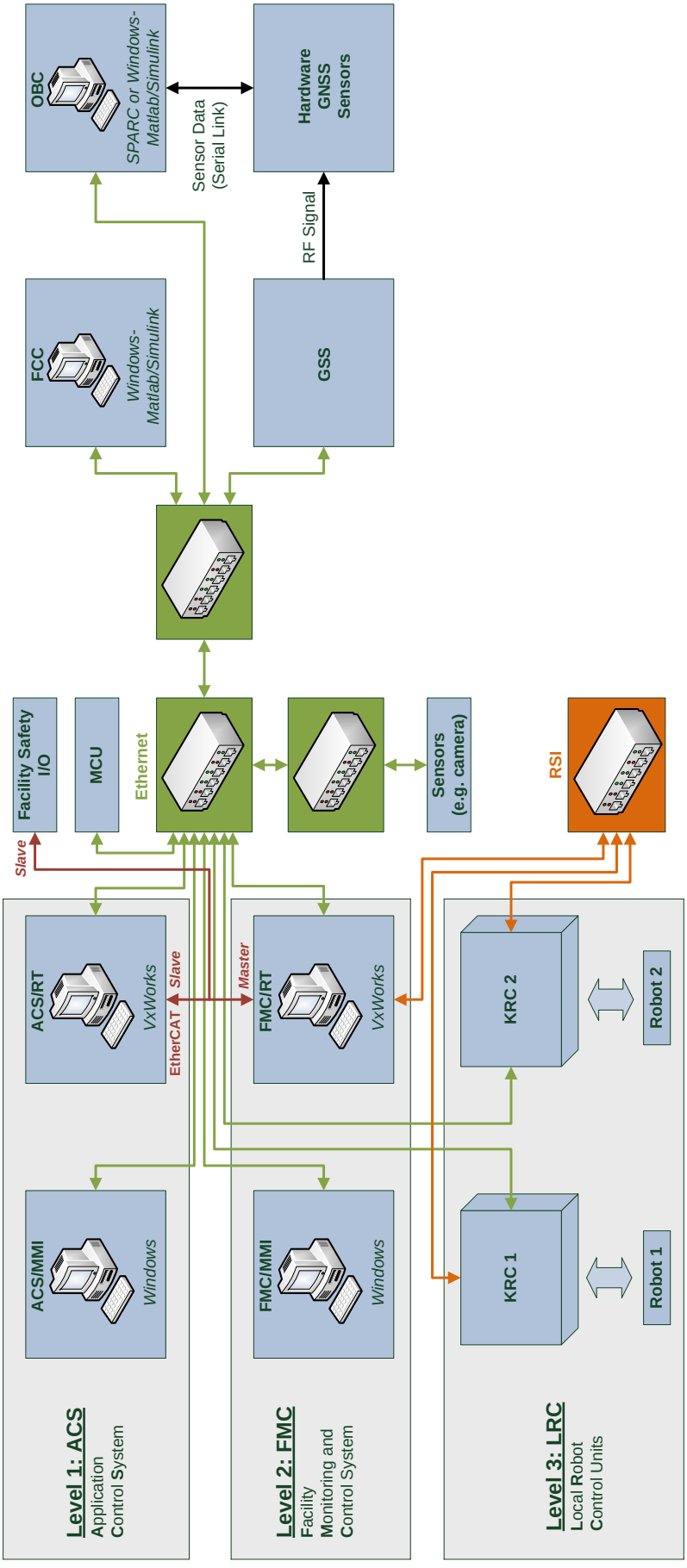

Figure 1.6.: EPOS Facility Control connected to the Formation Flying Testbed

- Since the ACS/RT uses the RTOS VxWorks combined with Matlab/Simulink Real-Time Workshop, the code has to be ported to this operating system (in the form of S-Functions). The compatibilty with VxWorks may require major changes to the code. Furthermore, Simulink RTW may not support S-Functions written in the programming language of the external simulator. Even if Simulink RTW could be modified to do so, most likely the effort necessary would be considerable. Another possibility: The simulator uses Simulink blocks which are incompatible with RTW or it may use a totally different simulation environment (e.g. Modelica). In that case the simulation would have to be designed anew from scratch.

- As mentioned before, the simulation system could include hardware connected to it, e.g. the Phoenix GPS receivers in the FF-Testbed. Then, interfaces which work together with VxWorks are necessary. Since I/O instructions are seldomly platform independent, most definitely these interfaces have to be redesigned to fullfill compatibility. Another problem considers the hardware side of interfaces. The ACS/RT has only a limited number of different types of interface connections available (ethernet, serial port, USB without drivers at the time this thesis is written). However, the simulation system may require other kinds of interfaces.

- The simulation on the ACS/RT has to run at a sample frequency of $250Hz$. This is a comparatively small sample time. Usually, GNC flight software (and associated simulators) run at $10Hz$ at most, just like the FF-Testbed. From this discrepancy follow several problems. It may not be possible to make the simulator run at that high rate due to the connected hardware components. Also, the required computation power to make the simulation work at $250Hz$ could exceed the ACS/RT's capabilities. EPOS requires a motion command for the robots every $4ms$ by design. At $10Hz$ the resulting "'jump"' in position and speed could overextend the robots' acceleration capabilities.

- While integrating their software, scientists would necessarily come in contact with EPOS internal software and Simulink blocks which are indispensable to the proper functioning of EPOS. On the one hand, this requires additional time and effort of the scienitsts. On the other hand, it might not be desirable to make each and every detail of EPOS know-how available to external customers.

It is the motivation of this thesis to avoid all these problems by designing a well-defined ethernet-based interface to connect any external simulation system to EPOS. This means developement and definition of a communication protocol as well as implementation of this protocol on the ACS/RT and, examplarily, an the FCC of the FF-Testbed.

Thus, the external simulation system can stay "'as it is"' and is simply connected to the EPOS network. Fig.1.6 shows the complete network configuration. No developement work is required, except the realization of the communication protocol. In most cases, small modifications to the code developed in this thesis for the FF-Testbed should be sufficient. In the worst case, this means new developement of an interface realizing the protocol for the specific platform. In any case, the work to be done is smaller by orders of magnitude compared to the problems outlined above. Ethernet is an established standard so that almost any conceivable simulation system is equipped appropriately. Moreover, the external simulator doesn't necessarily have to be located in the EPOS control room. It doesn't have to be in the same building and may even be in another city, as long as the ethernet link's latency is of reasonable magnitude.

There is one disadvantage which comes with ethernet. It is, by design, non-deterministic. The time some piece of information takes from source to target is of statistical nature. It is to be shown in this thesis to which extent this fact influences the proper functionality of the whole configuration.

1.5. Some Definitions

In this section, some definitions are given and general notation of variables, especially of those dealing with spatial movement, used throughout this theses is presented.

The author finds it helpful to make abundantly clear in which coordinate system some value is expressed and which coordinate frame serves as a reference (if needed). Therefore, the following notation is used in this thesis. As an example, an arbitrary variable like

$$\left(X_{sub1}^{super1}\right)_{sub2}^{super2}$$

is considered. The subscript $sub1$ denotes the "thing" the variable is associated with as well as other attributes. The superscript $super1$ corresponds to the coordinate frame the variable is expressed in, e.g. its components are expressed in. The variable may need a reference coordinate frame which the superscript $super2$ specifies. For example, angular velocity of a spacecraft may be the inertial angular velocity, e.g. ECI (Earth Center Inertial) is the reference frame ($super2$). However, the individual components of this angular velocity vector are expressed in body-fixed coordinates ($super1$). Note that wherever both $super1$ and $super2$ are the same, $super2$ is not shown at all to keep equations easily readable. This holds for the majority of cases. The subscript $sub2$ is used only at a few points in this thesis and may describe other attributes which considerably differ in type compared to $sub1$. Mainly, this notation is important for position, speed, attitude, angular velocity, simulation time steps and simulation times. For example

$$\boldsymbol{r}_{S/C1}^{ECI}$$

describes the position of some spacecraft S/C 1 expressed in ECI coordinates. No other subscripts or superscripts are required. The spacecraft's speed may be given by

$$\left(\boldsymbol{r}_{S/C1}^{CLW}\right)^{ECI}$$

which is expressed in CLW (Clohessy Wiltshire) coordinates and with ECI as the reference frame (inertial speed). A spacecraft's attitude can be expressed

$$\left(q_{S/C1}\right)^{ECI}$$

This is the spatial orientation (attitude) of S/C 1 (e.g. of S/C 1's body-fixed coordinate system) with respect to the ECI frame. There is no $super1$, since, from a logical point of view, there is no coordinate system the quaternion could be "expressed in". Beside this notation, another one is used for quaternions which specifically describe a coordinate transformation to put emphasis on the special function of the quaternion. A quaternion rotating from ECI to CLW my be written as

$$q^{ECI\rightarrow CLW}$$

Angular velocity is similar to speed. For example

$$\left(\boldsymbol{\omega}_{S/C1}^{CLW}\right)^{ECI}$$

may refer to angular velocity of S/C1 with respect to ECI, expressed in CLW coordinates. At this point, the following angular velocity is an example for utilization of $sub2$.

$$\left(\boldsymbol{\omega}_{S/C1}^{CLW}\right)_{n}^{ECI}$$

Here, the variable corresponds to a specific time step n of the simulation. In general, the simulation time of a Simulink model running on the ACS/RT will be different from the simulation time of a remote simulation (i.e. the FF-Testbed). The same point in time can be expressed in two different time frames just as position can be expressed in different spatial coordinate systems. Thus

$$t_{start}^{EPOS}$$

may be some kind of start time expressed in EPOS simulation time. Especially this notation of time frames is important when interconnecting two simulations and is therefore widely used in this thesis.

Another important issue is the angle convention associated with rotations. Here, given two coordinate systems A and B with coinciding origins but turned with respect to each other, A is turned

about axis $\mathbf{a}$ into B through *positive* angle φ which "points" from A to B. The quaternion associated with this rotation is written as the sum of scalar and vector part (see [10, p.157])

$$q^{A\to B} = \cos\left(\frac{\varphi}{2}\right) + \mathbf{a}\sin\left(\frac{\varphi}{2}\right) \tag{1.1}$$

Let $\mathbf{r}$ be some arbitrary vector, $\mathbf{r}^A$ its representation in A coordinates and $\mathbf{r}^B$ its representation in B coordinates. Then the expressions

$$r^B = \left(q^{A\to B}\right)^\star \, r^A \, q^{A\to B} \tag{1.2}$$

and

$$r^A = q^{A\to B} \, r^B \, \left(q^{A\to B}\right)^\star \tag{1.3}$$

hold ([10, p.157]). Here, r is the vector quaternion associated with the vector $\mathbf{r}$, that is

$$r = 0 + \mathbf{r} \tag{1.4}$$

In this thesis the author chooses to clearly distinguish vector and vector quaternion in order to achieve a consistent mathematical formulation. The (trivial) conversion is not carried out explicitly, rather the bold letter denotes the vector and its normal counterpart the vector quaternion.

2. Remote Control Architecture

2.1. Some Preliminaries

This chapter describes the whole EPOS remote simulation system from an abstract point of view. There are no concrete descriptions of implementation, although it's all about software, of course. (Only exception: Constants, i.e. packet types, are written as they appear in the code in order to avoid redundant and confusing designations for the same thing.) Rather, complying with a top-down approach, this chapter illustrates remote simulation architecture and breaks down all its key elements. All major equations are derived and all algorithms developed. Thus, this chapter alone allows to create *any* implementation of the remote simulation system which may be different from the author's implementation (see Ch.3) but compatible to it.

Although this chapter is of abstract character, it may be helpful to keep the general implementation principle in mind. The ACS/RT runs a Real Time Workshop generated Simulink simulation. Part of this simulation is the EPOS component of the remote simulation software (RemoteSim-Server) which is simply a custom Simulink block (S-function). On the other side, a Simulink simulation runs on a remote computer, for example the FF computer. This Simulink model contains the remote part of the remote simulation software (RemoteSim-Client). This again is nothing but a custom Simulink block (S-function). In conclusion, the software developed in this thesis is packaged in these two Simulink blocks.

Considering the step by step transfer of simulation data from the remote simulation to the EPOS CMD interface, there are several coordinate systems involved. They are outlined here at the beginning of this chapter to make understanding of subsequent explanations more straight forward. The user creates a Simulink model which contains a user-specific simulation connected to a RemoteSim-Client S-function block. The trajectory CMDs supplied to the client are expressed in a frame implicitly chosen by the user via the simulation model. This coordinate system is denoted User frame (U). RemoteSim may carry out a coordinate transformation from RemoteSim-Client to Server (and vice versa). The user may choose between two possibilities: Feedthrough and ECI to CLW conversion. The former comprises no conversion. Trajectory CMDs are given to RSP server outputs as they were sent by the client (interpolation added). The latter assumes that trajectory CMDs are given in ECI coordinates. They are converted to a coordinate frame which is located at the center of the target spacecraft. Its z-axis points towards the ECI origin, its x axis lies in the plane which is spanned by target spacecraft velocity vector and the aforementioned z-axis. The y-axis completes the right-handed-side coordinate system. In both cases, feedthrough and ECI to CLW, the coordinate system RemoteSim-Server's trajectory outputs are expressed in, are denoted CLW. Usually, the EPOS simulation will include another block which calculates the POV for the EPOS CMD interface such that the configuration of both spacecraft fits into the EPOS laboratory thereby "turning" and "shifting" the spacecraft while maintaining relative position and orientation. Therefore, the precise definition of CLW isn't critical. As long as CLW is a relative coordinate system, meaning that relative spacecraft positions, speeds, angular velocities and attitudes have reasonable values compared to robot performance, RemoteSim will work properly. The conversion carried out with ECI 2 CLW conversion by RemoteSim is only one possibility. If the user chooses feedthrough, the only condition is to supply CMDs with respect to a relative coordinate system leading to small values for the satellite states. In summary: The user commands in User frame (U) (which may be ECI or some relative coordinate system). RemoteSim-Server outputs these CMDs in CLW coordinates (either fed through from the Client or converted). These robot states are then converted via calculation of POV to match the robots' ranges. Note that for feedback of actual robot states the order of coordinate conversions is reversed, of course.

2.2. Excursion: Computer Networks

A central part in this thesis is computer networks. This section is intended to outline the main concepts and principles in computer networks so that the design choices made become clear and the developed protocols and their implementation can be clearly put into the context of computer networks. The subsequent deliberations are based on [7].

The EPOS global network shall serve as an example for a computer network. There are several computers, several switches and one router. Groups of computers are connected to the same switch. The switches are connected to the router. In network terms, computers are referred to as hosts. Moreover, hosts, switches and routers are all denoted nodes.

Usual computer networks use packet-based communication. This means that a certain quantity of information which constitutes a packet is transmitted from one end-point to another. The process of packet-switching allows different applications and different hosts to send their packets pseudo-simultaneously through the same cable.

In order to make it possible for the different hosts to "understand" each other the "'language"'used must be clearly defined and known to each host. Network protocols form the language in computer networks. A protocol defines the format of messages to be exchanged as well as the actions triggered by transmission and receipt of messages or by other events. Some of the network protocols are implemented in hardware and some are implemented in software.

Since the whole set of protocols is a very complex system, it is structured in a layer architecture compliant with a so-called service model. Each layer utilizes services provided by the layer below and provides services to the layer above on his part. The complete configuration of all layers constitutes a protocol stack. Computer networks can be structured as a five layer protocol stack ([7]). Fig.2.1 illustrates the protocol stack.

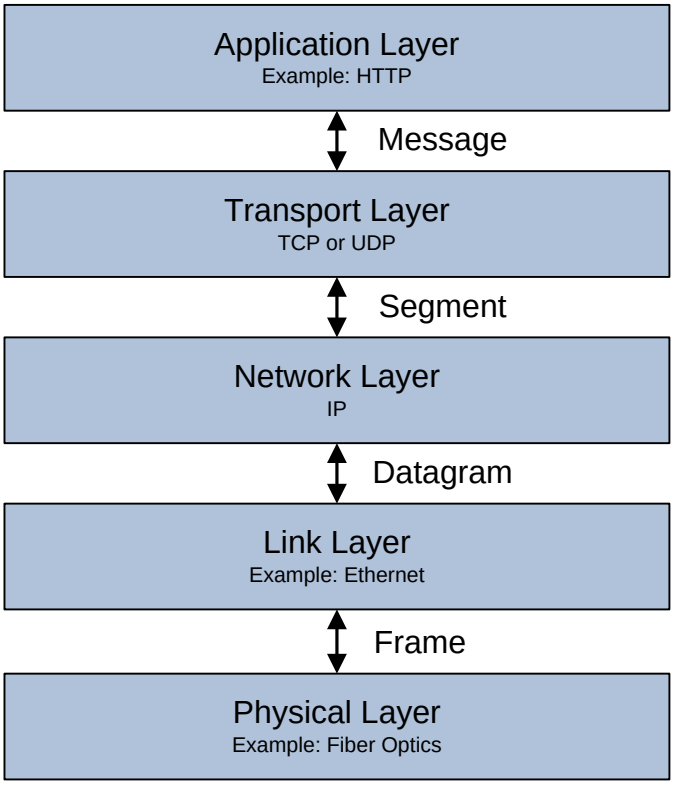

Figure 2.1.: Five Layer Protocol Stack

The five layers are briefly characterized from top to bottom.

- **Application Layer**: This is the top level protocol. It is implemented by an application. A good example for such an application is an internet browser which implements the Hypertext Transfer

Protocol (HTTP). The data packets sent and received by the application layer are denoted messages.

- **Transport Layer**: The transport layer protocol provides the service of exchanging application layer messages between (different instances of) applications, generally on different hosts. Thus, it provides a locigal connection between applications on hosts. An application is identified by a so-called port which is an integer ranging from 0 to 65535. There are two possible transport layer protocols: TCP and UDP. The data packets sent and received by the transport layer protocol are denoted segments.
- **Network Layer**: The network layer protocol provides the service of exchanging transport layer segments between different hosts. A host is identified by a so-called IP address which is a 32 bit number grouped into fours like 123.221.222.111. The data packets sent and received by the network layer protocol are denoted datagrams.
- **Link Layer**: The link layer protocol provides the service of exchanging network layer datagrams between different nodes. A famous example is ethernet where a node is identified by a Media Access Control Address (MAC). Note that not all nodes have to have a MAC address. For example, switches which simply pass through the received data packets do not need to be addressed by the link layer. The data packets sent and received by the link layer are denoted frames.
- **Physical Layer**: The physical layer protocol provides the service of exchanging the individual bits of a link layer frame between different nodes. Examples: Twisted pair, fiber optics.

If an application wants to transmit some message to another host, it has to pass the destination application port as well as the IP address of the host the target application runs on to the transport layer. Not only destination port and IP address are transmitted, but also source port and IP address, so that the destination application can identify the source of the message.

During transmission, a message passes the protocol stack from top to bottom at the source and then from bottom to top at the destination. The message is thereby encapsulated from layer to layer. The transport layer adds to the application layer message a header which is needed by the destination transport layer. Message and transport layer header constitute a transport layer segment. The network layer adds to the transport layer segment the network layer header resulting in a network layer datagram. The link layer adds its header to the network layer datagram and passes this link layer frame to the physical layer. At the destination host the different layers unpack the packet from bottom layer to top layer until the orginal message reappears.

In the terms of computer networks, this thesis is to develop and define an application layer protocol (which may be comprised of several "'subprotocols"') with associated messages, thus realizing all communication, processes and actions necessary to make a remote simulation work with EPOS. Farther, this also means to choose a transport layer protocol, TCP or UDP. Therefore these protocols are described briefly.

- **Transmission Control Protocol (TCP)**: TCP is a connection-oriented protocol. There must be a "'handshake"' between client and server to establish a TCP connection prior to exchanging segments. After communication is finished, the connection must be closed. The Transport Layer Protocol provides reliable data transfer. It garantess that a message sent is transported completely to its destination. Furthermore, it garantees that messages are received in the same order they were sent. It provides to the application layer the service of transmitting and receiving a continuous stream of bytes. Finally, TCP includes a congestion-control mechanism which throttles data rate if the network is congested and collects small quantities of data before it transmits it to the destination. This congestions mechanism is unfavorable for real-time applications like the one developed in this thesis. For such purposes it is vital that data is sent instantly to keep latency as low as possible. Fortunately, this mechanism can be switched off programmatically.
- **User Datagram Protocol (UDP)**: UDP is a minimum transport layer protocol. It therefore lacks many services provided by TCP. UDP is connection-less. There is no handshake and messages are sent without establishing a connection. They are thus received "as they come".

The protocol is unreliable. There is no garantee that a message is transmitted successfully to its destination or that no message gets lost in the network. In contrast to TCP, where a stream of bytes can be sent, UDP provides to the application layer only the service of transmission and receipt of finite packets of data. Also, there is no congestion-control. As a result, UDP is faster than TCP due to the lack of more complex services and segment overhead.

2.3. EPOS Remote Control Requirements

This section is intended to give an overview of the main problematic areas that have to be considered in designing a remote simulation architecture for EPOS. Fig.2.2 shows requirements as a SysML diagram.

There are two great issues to deal with. First, the interaction between the remote simulation and EPOS has to be managed. Second, an appropriate data connection has to be supplied. Since the former relies on the latter, these tasks well be treated separately in this section, beginning with simulation interaction.

The procedure for synchronous commanding (see 1.2.4) has to be dealt with. It is labeled **Facility Initialization** in Fig.2.2. Intial positions and attitudes have to be provided to the facility so that *Move to Start* can be carried out properly. Note that these initial conditions are merely for allowing to enter synchronous commanding mode. The Initial conditions of the actual remote simulation have to be taken care for at a different stage. Note also that the facility initial conditions assume zero speed and angular velocity whereas those of the remote simulation do not.

The requirement **Synchronous Timing** addresses the need for having some measure of synchronization between remote simulation and EPOS. The fact alone that ethernet is used obstructs any real-time synchronization. Rather, this requirement demands a certain interconnection. Move commands are not to be executed simply when they are received but at well-defined periodic time steps. The remote simulation has to send these CMDs in time for the right time step. This implicitly encompasses several tasks.

Since the sample frequency of the ACS/RT simulation is $250Hz$ (by design) and the remote simulation most likely runs at a much lower frequency (FCC of FF-Testbed: 1 to $10Hz$), robot positions and attitudes have to be interpolated in-between CMDs. In order to avoid infinite acceleration, which would have the facility to stop the simulation for safety, the robots' speeds and angular velocities have to be continuous. This is equivalent with a C^1 continuous **Interpolation**.

Ethernet is non-deterministic. Therefore it is conceivable that a move CMD is sent too late occasionally on a stastical basis. To consider this there must be a possibility for **Extrapolation**. Of course, extrapolation only makes sense for a fraction of the time between move CMDs. Similar to interpolation, extrapolation must be C^1 continuous.

Seamless Transition from Initial Conditions to Simulation is imperative. While the robots move such that they reach its initial positions and attitudes with initial speeds and angular velocities, the remote simulation has to wait. Only shortly before the initial states are reached, the remote simulation has to start sending move CMDs such that temporal interconnection is intact.

Finally, it is necessary to **Monitor Timing**. Thus, a temporal drift between remote simulation and EPOS ACS/RT can be detected and handled.

As mentioned before **Realization of Starting Conditions** plays an important role and is one of the advantages of using a remote simulation with EPOS. As mentioned above, the initial conditions associated with facility initialization are restricted to zero speed and angular velocity. This requirement ensures that a simulation may start with arbitrary speed and angular velocity (within the limits of facility performance).

Reentrancy is another important issue. Restarting the simulation on the ACS/MMI comes with rebuilding of the code, connecting to the ACS/RT and new facility initialization procedure. The remote simulation concept is to avoid this necessity. With the facility intialized and the ACS/RT running in synchronuous mode, the remote simulation can be started and stopped as the user pleases. This implies that a broken link is detected and the robots are slowed down safely. In that state a restart of the remote simulation is possible.

Closely connected to reentrancy is to **Limit Robots' Speed and Acceleration**. If the robots' performance limits are violated the facility stops the simulation. This conflicts with reentrancy re-

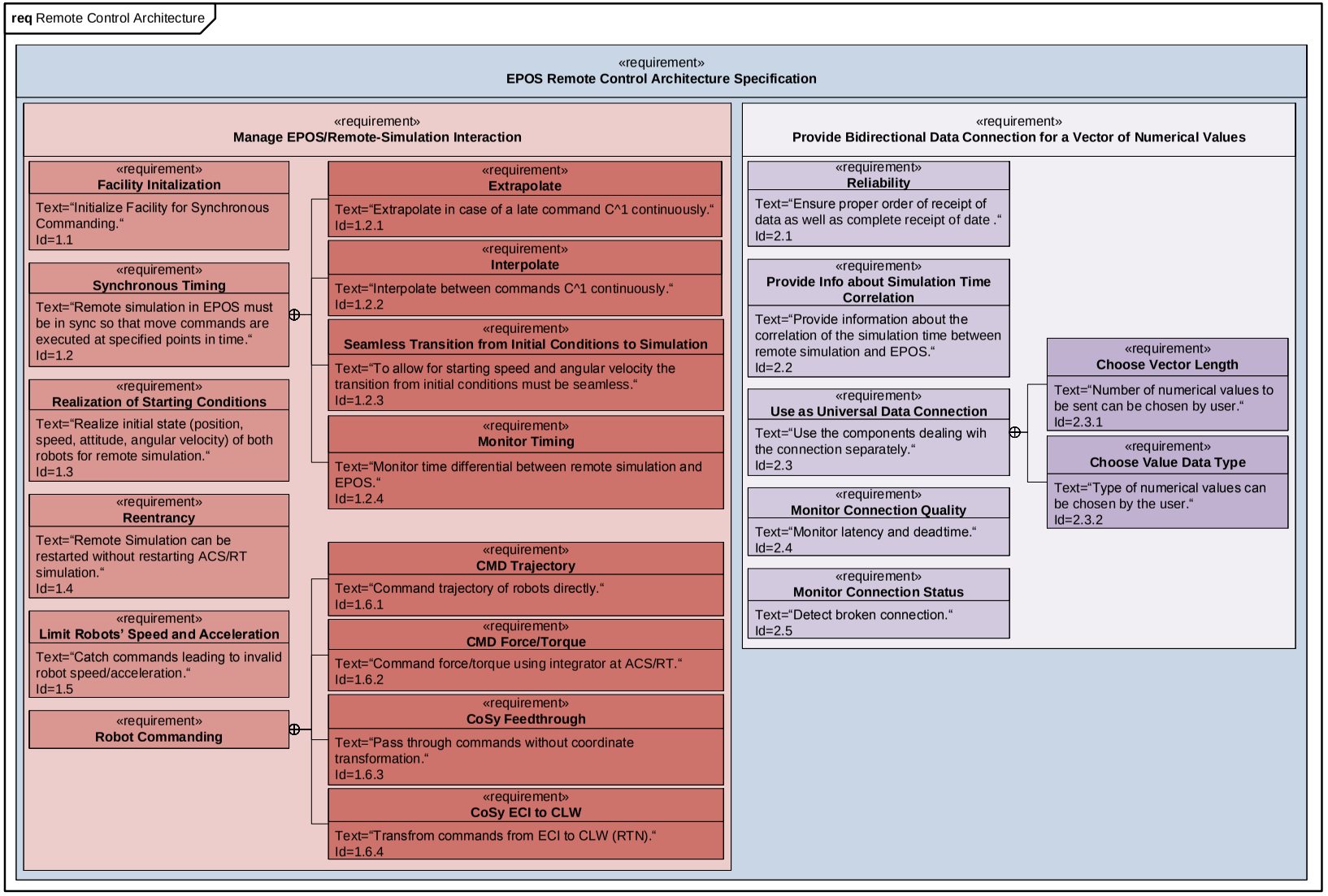

Figure 2.2.: SysML Requirements Diagram of Remote Control Architecture Requirements

quirement. Moreover, it is a matter of safety to limit the robots' movements considering that control is given to an external source, beyond the EPOS laboratory.

It must be possible to carry out **Robot Commanding** in different ways. In the simplest form, **CMD Trajectory**, the robots states are sent by the remote simulation. This means that any orbit model (including the integrator of orbital mechanics and rigid body dynamics) runs in the remote simulation (with according low sample frequency). However, it may be advantageous to have the integrator run on the ACS/RT and exploiting the high symple frequency. In this case, force and torque is commanded rather than S/C states: **CMD Force/Torque**. Both modes of operation can be used with or without coordinate transformation. With **CoSy Feedthrough** no coordinate transformation is applied and the commands appear at the ACS/RT as the are sent by the remote simulation. With **CoSy ECI to CLW** it is assumed that CMDs are sent in ECI coordinates which are then transformed to CLW coordinates at the ACS/RT.

The second great requirement is **Provide Bidirectional Data Connection for a Vector of Numerical Values**. Any process and action of the remote simulation architecture relies on the connection.

Speed and low latency of data transfer cannot be considered a requirement, since they are given by the available physical hardware (ethernet, switches etc.) and cannot be influenced by the author. However, choice of transport layer protocol and design of application layer communication mechanisms can influence **Reliability**. It is imperative that commands are received in the same order as they were sent. Moreover, no single command shall be lost in the network. A review of computer network principles (Sec.2.2) proofs these requirements not to be taken for granted.

As stated above, remote simulation and EPOS have to be temporally interconnected during a simulation run. This interconnection is possible, only if there is some kind of information about both simulation times. This need is to be satisfied by the requirement **Provide Info about Simulation Time Correlation**, with the exact nature of this information yet to be determined.

In order to detect communication problems, the system shall be able to **Monitor Connection Quality**. The primary aim is data latency rather than data througput. Thus, it is possible to make a statement about how long data takes from transmission by the remote simulation to receipt by EPOS and back again.

Another requirement related to reentrancy: **Monitor Connection Status**. It must be possible to detect if a link isn't intact any more.

An application standing somewhat aside from EPOS remote control is general data exchange between two simulations. The components dealing with the pure data connection shall also work stand-alone to **Use as Universal Data Connection**. As such, it isn't sufficient to make a single configuration available. Rather, the number of numerical values to transmit and receive per time step must be variable as well as the data type of the values. In a stand-alone application the user is to **Choose Vector Length** and **Choose Value Data Type**. The idea behind this additional requirement is based on the need to transfer general sensor data (e.g. a camera picture) between simulations or for monitoring purposes. Also, this implies a certain modularity in design of remote control architecture.

2.4. Two Layers - SimCon and RemoteSim

The nature of the various tasks which have to be dealt with in order to realize a connection between some remote simulation and EPOS suggest a splitting into two parts or layers. Here, the author complies with the useful and widespread concept of layers in software design, outlined in Sec.2.2. Fig.2.3 illustrates the two layers, their connection and their roles in remote control architecture.

The Simulation Connection Protocol (SCP) constitutes the bottom layer. The implementation of this protocol is also denoted SimCon. The Remote Simulation Protocol (RSP) constitutes the top layer. The implementation of this protocol is also denoted RemoteSim.

SCP provides a bidirectional connection of "raw" data in the form of a vector of numerical values. Data type of these values and vector length can be chosen. The format in which this data is exchanged is specified as a SimCon packet. Moreover, SCP monitors connection quality (packet latency, deadtime etc.) and signal status (RSP link intact or broken) and provides this information to RSP. It also supplies the top layer with information concerning the correlation between EPOS' simulation time and the remote computer's simulation time. This is necessary to realize proper timing between both remote simulation end points. Thus, SCP can be said to manage the actual data connection and associated low-level functions.

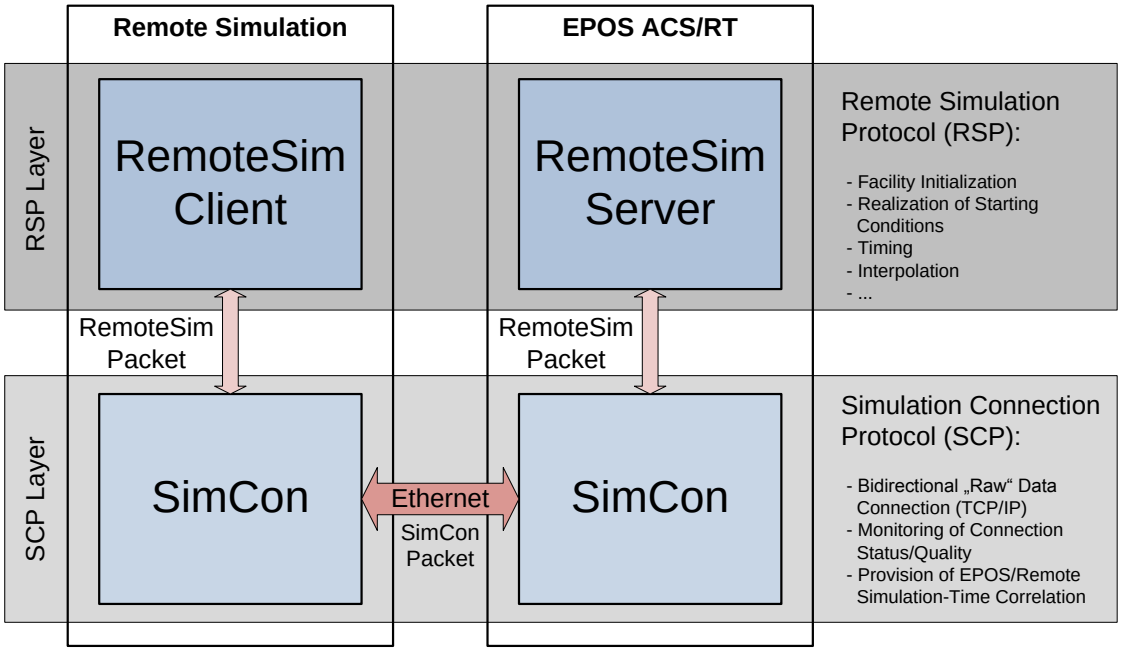

Figure 2.3.: Top Remote Control Software Architecture

RSP deals with the different states of the simulation process, i.e. facility Initalization, realization of starting conditions, seamless transition to actual simulation etc.. It handles many other tasks, interpolation and timing being only two of them (see Sec.2.6). RSP utilizes the services provided by SCP. RemoteSim communicates using RemoteSim packets which are comprised of a number of double values, each having a well-defined meaning. Since a RemoteSim packet is nothing but a data vector of numerical values, SCP can be used to exchange these packets between remote simulation and EPOS ACS/RT. Thus, RSP can be said to manage the interaction between remote simulation and EPOS.

As depicted in Fig.2.3, the remote simulation comprises one instance of SimCon and one of RemoteSim-Client. On the EPOS ACS/RT one instance of SimCon and one of RemoteSim-Server is required. Note that at this abstract level both are separate components and can also be at the code level. However, both may be combined in one Simulink block in the remote simulation and one block on the ACS/RT.

This partitioning of tasks and responsibilities has several advantages. SCP, i.e. SimCon can be used separately as a universal data connection. It may transmit sensor data from simulation to simulation like visual information or something similar, thus complying with requirement **Use as Universal Data Connection** (see Sec.2.3). Furtheremore, RemoteSim could be combined with a different data exchange protocol/software as long as it provides the services RSP requires.

In the terms of computer networks, SCP and RSP comprise the application layer protocol, as illustrated by Fig.2.4. A RemoteSim packet is encapsulated in a SimCon packet which then constitutes a network message.

2.5. Simulation Connection Protocol (SCP)

2.5.1. Choice of Transportation-Layer Protocol

As a first step the kind of physical link used by SCP has to be selected. Due to a number of reasons ethernet LAN is given as a boundary condition. It is already available in the form of the EPOS local ethernet network. Moreover, it is relatively easy to use from a programmer's perspective. Using ethernet, it is also possible to communicate over the internet with a computer farer away than within the boundaries of the EPOS laboratory, which allows for possible future extensions. And the most significant argument: No matter the platform of a dynamic simulation to be connected to EPOS, it

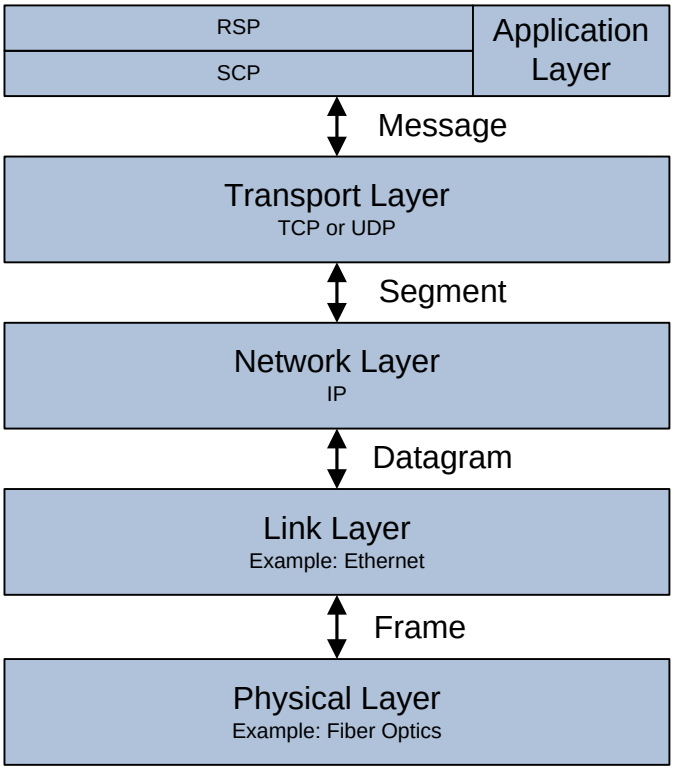

Figure 2.4.: RSP and SCP in Protocol Stack

is very likely that ethernet is supported. All these advantages outway the drawback of non-realtime behaviour of ethernet. There is no garantee that some piece of data will be transmitted withing a certain amount of time. Using a special real-time bus would obliterate all the above mentioned advantages.

Another decision to be made beforehand is the choice of transport-layer protocol. There are two possiblities TCP or UDP. Both are briefly described in Sec.2.2. Considering the requirements of reliable data transfer (see Sec.2.3) TCP is the obvious choice. It garantees the right order of data transfer and it makes sure that no data is lost in the network. Thus, these two requirements are already fullfilled and must not be taken into account in software design.

2.5.2. SCP Parameters, Inputs and Outputs

SCP requires several parameters for operation. Tab.2.1 gives an overview with name, symbol and a short description.

Table 2.1.: Parameters of SCP

SCP_PARAM_SIMCON_TYPE	-	Valid values: SIMCON_TYPE_CLIENT or SIMCON_TYPE_SERVER. Two instances of SCP can communicate, if one is of server on the other of client type.
SCP_PARAM_DATA_TYPE	-	The data type of the numerical values to be transmitted or received. Valid values: SIMCON_PORT_TYPE_INT8, SIMCON_PORT_TYPE_UINT8 and SIMCON_PORT_TYPE_DBL64.
SCP_PARAM_SEND_VEC_LENGTH	-	The length of the vector to be transmitted.

Continued on next page

Table 2.1 – *Continued from previous page*

SCP_PARAM_RECV_VEC_LENGTH	-	The length of the vector to be received. It must match the length of the vector sent by the other SCP instance for proper communication.
SCP_PARAM_CONTROL_PORT	-	TCP port for the SCP control connection. Must be the same with both communicating instances of SCP.
SCP_PARAM_DATA_PORT	-	TCP port for the SCP data connection. Must be the same with both communicating instances of SCP.
SCP_PARAM_IP	-	IP address of the machine to communicate with.
SCP_PARAM_TIMEOUT	Δt_{ping}	Timeout in $[ms]$ for determining a broken link. This is the time interval SCP waits for the return of a sent ping packet.

SCP has several inputs. Tab.2.2 gives an overview with name, symbol and a short description.

Table 2.2.: Inputs of SCP

SCP_IN_DATA_INT8 SCP_IN_DATA_UINT8 SCP_IN_DATA_DBL64	The vector of numerical values to be transmitted. The proper input to choose depends on SCP_PARAM_DATA_TYPE. If the wrong input is chosen, the data won't be transmitted.

SCP has several outputs. Tab.2.3 gives an overview with name, symbol and a short description.

Table 2.3.: Outputs of SCP

SCP_OUT_DATA_INT8 SCP_OUT_DATA_UINT8 SCP_OUT_DATA_DBL64	- - -	Returns the received data vector of numerical values. The proper output depends on SCP_PARAM_DATA_TYPE as well as the data type of the received values. Both must match.
SCP_OUT_TARGET_REF_TIMESTEP SCP_OUT_HOME_REF_TIMESTEP	n_{ref}^{T} n_{ref}^{H}	Reference time steps of home and target instance of SCP. The combination of both (with given home and target sample time) determine the correlation between home and target time steps.
SCP_OUT_CONNECTION_OK	-	Indicates that a SCP connection is intact.
SCP_OUT_IS_NEW	-	Indicates that a new data vector has been received and has not been read yet.

Continued on next page

Table 2.3 – *Continued from previous page*

SCP_OUT_SEND2RECV	Δt_{s2r}	These outputs allow to monitor signal quality as described in Sec.2.5.7. In addition to the value itself, the outputs also comprise mean, noise, jitter, standard deviation and signal-to-noise ratio.
SCP_OUT_ACQU2OUTPUT	Δt_{a2o}	
SCP_OUT_DEADTIME_SEC	Δt_{dead}	
SCP_OUT_DEADTIME	Δn_{dead}	

2.5.3. General Properties

SCP uses TCP/IP as transport- and network layer protocol. A SimCon link uses two ports for two associated connections: a data connection and a control connection. As the terms suggest, data is sent via data connection and control information is sent via control connection. Thus SCP can be said to send control information "out-of-band", similarly to the File Transfer Protocol (FTP) and the Real-Time Streaming Protocol (RTSP) (see [7, p.141]). The reason for this is simple. Assume that a large data vector is sent. While it is transmitted, no other information can be exchanged. Using another connection specifically for control information obliterates this drawback. In a way, the multiplexing/demultiplexing algorithm of the underlying TCP is exploited to send control information and data vectors at the same time (see [7, p.227]). SimCon always establishes a point-to-point link between two SimCon instances. One instance is a server, the other is a client. A server accepts only one client at the same time. Behaviour of client and server are similar, besides the procedure of connection establishment, as dictated by TCP. Information is transmitted in the form of packets with a well-defined structure and of different types. Packets of type data are sent using the data connection, all other types of packets are sent via control connection. A packet received at the wrong port is discarded. The following section shows the packet structure in detail.

2.5.4. SCP Packet Structure

Fig.2.5 shows the structure of a SimCon packet. The partition in lines is only for achieving a good overview. The numbers an the beginning of the lines indicate the index of the byte the line starts with.

The beginning of a SimCon packet is marked by two constant bytes, a 0x00 and 0x02, followed by the packet header. The header begins with the overall size of the packet. This is an 32-bit unsigned integer. Then one 8-bit unsigned integer indicates the version of SCP. This header field allows for a future version of SimCon, which may make use of additional header fields etc.. The version of SCP developed in this thesis doesn't use this header field yet. Another unsignend integer of one byte length represents the packet type. A thorough presentation of the different SCP packet types will be given below.

The succeeding eight values are required for measuring and monitoring dead-time and packet delay. They are grouped in two similar sections. One numbered 0 and the other numbered 1. This indexing has the following background. 0 referes to header fields that are associated with the present SCP packet (in a sense "t_0"). The other index referes to the SCP packet that had been received shortly before the present packet was sent by the other SCP instance (sse the next section for a detailed explanation of this mechanism). So the index 1 referes to a time "t_{-1}". To have these header fields make sense, the individual instance of SimCon must have a time index in seconds available for calculations. The offset of this time is not important. It might be the time since the computer was booted or since the simulation was started. Furthermore, the time indices of the two computers communicating with each other don't have to be equal.

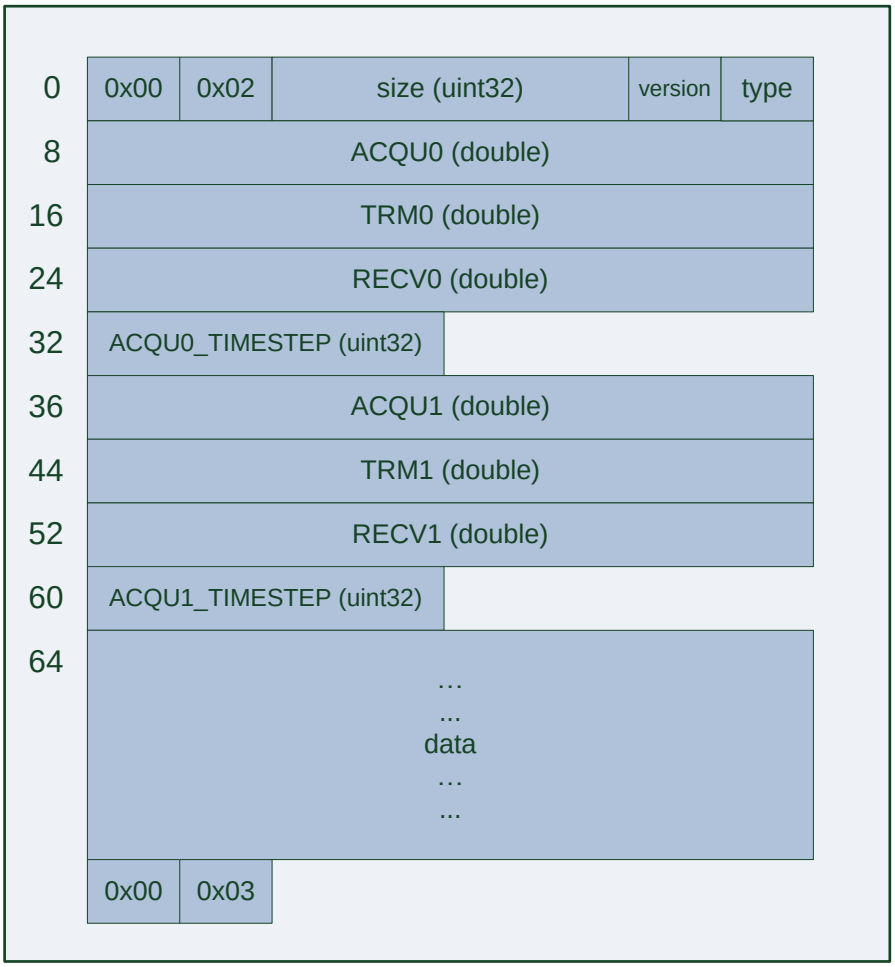

Figure 2.5.: SimCon Packet Structure

2.5.5. SCP Packet Exchange Process - Forward and Return Packets

In order to understand the SimCon packet header fields it is useful to look at the relation of transmitted simulation data to a target instance of SimCon and its response data back to the home instance. As an example, suppose a simulation is split up into the dynamic behaviour of a satellite on the home computer and some kind of controller on the target computer. At some simulation step the home computer forwards a data packet containing the satellite's state to the target computer. Denote this packet the forward packet. The controller on the target computer calculates control forces and torque for the satellite and sends them back in another data packet. Denote this packet return packet. When the forward packet has arrived at the target simulation (controller) its header fields ACQU0, TRM0, RECV0 and ACQU0_TIMESTEP have the following meaning.

- ACQU0: The time when the data contained in the forward packet was acquired from the simulation on the home computer simulation, expressed in home time frame, in seconds.
- TRM0: The time when the forward packet was actually sent by the home computer, expressed in home time frame, in seconds.
- RECV0: The time when the forward packet was received by the target computer, expressed in target time frame, in seconds.
- ACQU0_TIMESTEP: This time is somewhat similar to ACQU0. However, it is not given in seconds but rather in time steps. Here it is assumed that both simulations on target and home computer work with a fixed step size. Each simulation step can be associated with an index.

This header field represents the value of this index at the time of the packet data's acquisition from the simulation on the home computer.

The forward packet's data is given to the simulation (controller) on the target machine. Its response is given the the target computer's SimCon instance which creates the return packet. Now the header fields of the forward packet (ACQU0, TRM0, RECV0 and ACQU0_TIMESTEP) are copied to the header fields ACQU1, TRM1, RECV1 and ACQU1_TIMESTEP of the return packet. While the return packet's data is acquired and sent by the target computer and subsequently received by the home computer, its header fields ACQU0, TRM0, RECV0 and ACQU0_TIMESTEP are filled similarly to those of the forward packet. (Except home and target computer are switched in the list above.) Keeping the concept of forward and return packet, the header fields of the return packet have the following meaning.

- ACQU0: The time when the data contained in the return packet was acquired from the simulation on the target computer simulation, expressed in target time frame, in seconds.
- TRM0: The time when the return packet was actually sent by the target computer, expressed in target time frame, in seconds.
- RECV0: The time when the return packet was received by the home computer, expressed in home time frame, in seconds.
- ACQU0_TIMESTEP: The time step when the data contained in the return packet was acquired from the simulation on the target computer's simulation, expressed in target time frame, in simulation steps.
- ACQU1: The time when the data contained in the forward packet was acquired from the simulation on the home computer simulation, expressed in home time frame, in seconds.
- TRM1: The time when the forward packet was actually sent by the home computer, expressed in home time frame, in seconds.
- RECV1: The time when the forward packet was received by the target computer, expressed in target time frame, in seconds.
- ACQU1_TIMESTEP: The time step when the data contained in the forward packet was acquired from the simulation on the home computer simulation, expressed in home time frame, in simulation steps.

Fig.2.6 illustrates the meaning of these header fields. There are two timelines, one for the home computer and one for the target computer. Simulation steps are indicated by small vertical black lines. t_i^T denotes the time step i of the target simulation and t_j^H the time step j of the home simulation, respectively. When a calculation step is to be carried out, data is acquired. Shortly after that, the associated packet is transmitted. The transmission takes some time of course, hence the blue lines are not vertical but inclined. Finally, there also is a time interval between receipt of a packet and the next simulation step. As a result, t_{recv1} and t_{acqu0} differ in general.

It should be noted that there is not always a return packet associated with a forward packet or vice versa, at least if the packets are of type data. Consider a case where the sample time of the target machine is much larger then the sample time of the home machine. This means that the data of a forward packet received by the target computer might become outdated by a succeeding forward data packet, if more than one forward data packet arrives before the next simulation step is due. SCP dictates that the "'old"' data be discarded in such a case. Thus it is always the most recently received data the simulation is provided with. In the same manner, the header fields ACQU1, TRM1, RECV1 and ACQU1_TIMESTEP of a return packet are associated with the most recently received forward packet.

Note also that the definition of forward packet, return packet, target computer and home computer depend on the point of view. They are used here for illustrative reasons. Of course, any packet can be considered to be a forward packet or a return packet, respectively. Subsequently, it will be used in this thesis, since a clear assignment of forward and return packet clarifies the relation of header values automatically.

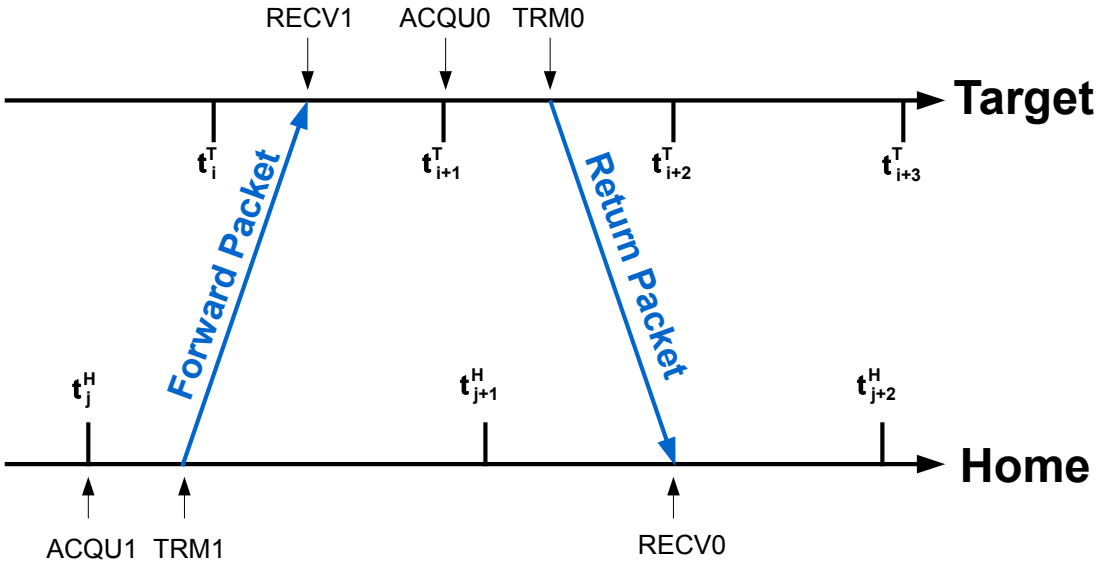

Figure 2.6.: Meaning of Time Header Fields

2.5.6. SCP Packet Types

SCP knows different types of packets distinguished by the header field "'type"' contained in each SimCon packet. Here, an overview is presented, while Sec.2.5.7 discusses SCP behaviour and thereby gives a more thorough description of the packets' purpose. No matter the packet type, a SimCon packet's header fields are always to be set as outlined in Sec.2.5.5. The following overview also indicates the return packet a forward packet has to be associated with.

Type: `PACKET_TYPE_DATA_DBL`, `PACKET_TYPE_DATA_INT8`, `PACKET_TYPE_DATA_UINT8`
Connection: data
Description: SimCon data packet. The values contained in the data section of the packet represent the vector of values to be transmitted from one simulation/computer to another. (E.g. state vector, control forces ...) The packet type also specifies the values' datatype. If sent, this packet type can always be considered to constitute a return packet with the associated forward packet beeing the most recently received data packet.

Type: `PACKET_TYPE_PING`
Connection: control
Description: This kind of packet contains no data. It is simply a request directed towards the target computer to return an according `PACKET_TYPE_PING_RETURN` packet to confirm that the SimCon link is intact. `PACKET_TYPE_PING` is the forward packet, `PACKET_TYPE_PING_RETURN` the return packet.

Type: `PACKET_TYPE_PING_RETURN`
Connection: control
Description: This kind of packet contains no data. It is the response to the receipt of a ping packet.

Type: `PACKET_TYPE_DELAY`
Connection: control
Description: This kind of packet contains no data. It is a request directed towards the target computer to return a `PACKET_TYPE_DELAY_RETURN`. Usually it is sent in regular time intervals, independently from time steps. `PACKET_TYPE_PING` is the forward packet, `PACKET_TYPE_PING_RETURN` the return packet.

Type: PACKET_TYPE_DELAY_RETURN
Connection: control
Description: This kind of packet contains no data. It is the response to a PACKET_TYPE_DELAY packet. It is used to make an estimate on the offset between the time index of the target computer and the time index of the home computer.

Type: PACKET_TYPE_REF
Connection: control
Description: This kind of packet contains no data. However, its ACQU0_TIMESTEP field combined with the current timestep of the receiving SimCon instance can be used to determine the correlation between home and target simulation times. To ensure the determination of this correlation, a PACKET_TYPE_REF has to be sent at any timestep. There is no associated return packet.

2.5.7. Behaviour

This section defines the behaviour of SCP. The author finds it convenient to structure this description according to the different types of packets.

First of all, there is no handshake or similar connection establishement procedure on SCP level. As soon as a link is established (both data and control) packets can be sent and received. Moreover, a link can be "killed" without notifying the communication partner. The "ping" mechanism (see below) allows for detecting a broken link.

A packet of type PACKET_TYPE_PING can be sent anytime via control connection. Packets of this type are to be sent at regular intervals. Upon receipt of a packet of type PACKET_TYPE_PING, SCP dictates to expidetiously respond with a PACKET_TYPE_PING_RETURN. This mechanism, e.g. the combination of these two packet types can be used to check for an intact link. If a PACKET_TYPE_PING_RETURN packet is not received as a response to a PACKET_TYPE_PING packet before a certain timeout, the link can be considered broken and according measures can be taken. The header fields ACQU0, TRM0, RECV0, ACQU1_TIMESTEP, ACQU1, TRM1, RECV1 and ACQU1_TIMESTEP are not specified for these packet types.

A packet of type PACKET_TYPE_DELAY can be sent anytime via control connection. This is optional. SCP does not ask for use of this kind of packet obligatorily. The fields TRM0 and RECV0 have to be filled during transmission by home computer and receipt by target computer as described in Sec.2.5.5. When the target computer receives the packet, SCP dictates that it must be responded by returning a packet of type PACKET_TYPE_DELAY_RETURN via control connection expidetiously. The fields TRM1 and RECV1 are obtained from the fields TRM0 and RECV0 of the PACKET_TYPE_DELAY packet. The fields TRM0 and RECV0 of the PACKET_TYPE_DELAY_RETURN packet are determined as given in Sec.2.5.5. ACQU0, ACQU0_TIMESTEP, ACQU1 and ACQU1_TIMESTEP are indeterminate and have no meaning. The header fields of a received PACKET_TYPE_DELAY_RETURN can be used to calculate the reference time between the time frames of home and target computer. As mentioned before, the time indices used to fill the packets' header fields most likely are not and don't have to be in sync. But in order to use the header fields of a packet (for a example a data packet) to make a statement about the delay or dead-time there must be some kind of information how the time frame of the target computer simulation is related to that of the home computer simulation. For that purpose SCP defines a time transformation $t^{T\to H}$.

$$\boxed{t^{T\to H} = t^H - t^T} \tag{2.1}$$

where t^H denotes an arbitrary point in time in the home computer simulation time frame and t^T denotes the *same* point in time in the target computer simulation time frame. Thus, from the perspective of the home instance of SimCon, a time can be transformed from the target computer time frame using the equation

$$t^H = t^{T\to H} + t^T \tag{2.2}$$

How is a PACKET_TYPE_DELAY_RETURN used to determine this transformation? Given the header fields of the packet, let's assume that the time it took to transmit the forward packet equals that to transmit the associated return packet.

$$t^H_{recv1} - t^H_{trm1} = t^T_{recv1} - t^T_{trm1} = t^T_{recv0} - t^T_{trm0} = t^H_{recv0} - t^H_{trm0} \tag{2.3}$$

Now we need the same point in time expressed in the target time frame and in the home time frame. Denote a point in time just in the middle between transmission and receipt reference time.

$$t^T_{ref} = 0.5 \cdot (t^T_{trm0} - t^T_{recv1}) + t^T_{recv1} \tag{2.4}$$

$$t^H_{ref} = 0.5 \cdot (t^H_{recv0} - t^H_{trm1}) + t^H_{trm1} \tag{2.5}$$

The time transformation can then easily be calculated.

$$t^{T \to H} = t^H_{ref} - t^T_{ref} \tag{2.6}$$

Fig.2.7 illustrates the relationship between the header fields and reference times.

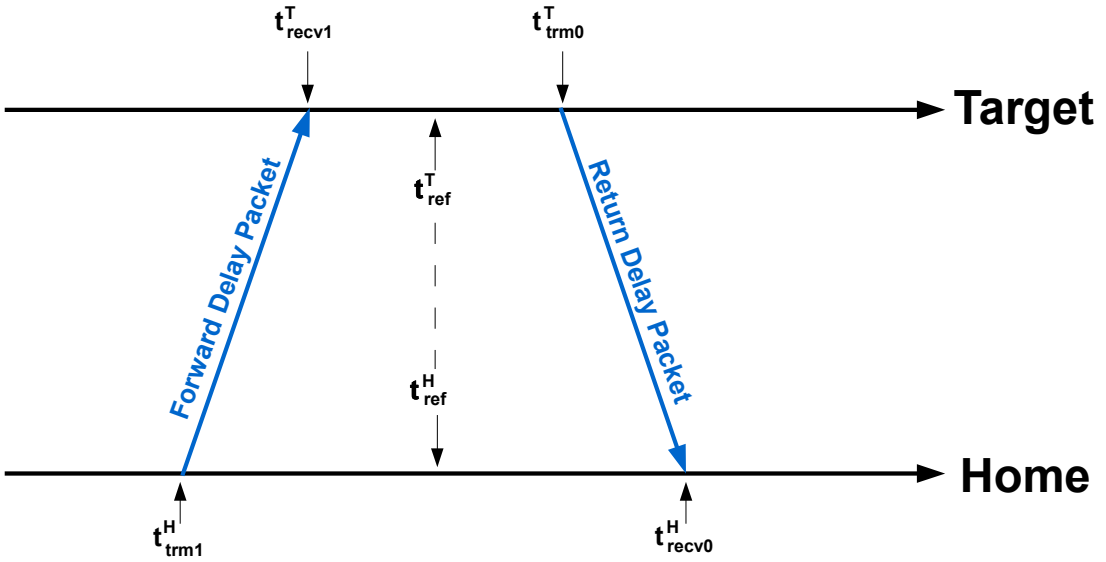

Figure 2.7.: Calculation of Time Transformation

A PACKET_TYPE_DELAY_RETURN packet contains all the information needed for determining this time transformation. Note that the assumption (2.3) won't be fullfilled generally. Within the boundaries of a local network it may be sufficiently accurate, in the internet it may be hard to fullfill. Therefore, it may be necessary to use a series of PACKET_TYPE_DELAY_RETURN packets and calculate an average time transformation. The author intends to keep the SimCon Protocol independent from any implementation. That's why SCP doesn't define such an averaging mechanism.

A data packet (PACKET_TYPE_DATA_DBL, PACKET_TYPE_DATA_INT8...) can be sent any time. Its header fields (ACQU0, TRM0, RECV0, ACQU0_TIMESTEP, ACQU1, TRM1, RECV1, ACQU1_TIMESTEP) must be filled according to Sec.2.5.5. Upon receipt of a data packet SCP dictates that its type, e.g. the datatype of the values contained in the packet's data section, be consistent with the SimCon instance's configuration. If this isn't the case, the packet will be discarded. The aforementioned header fields can now be uilized to make a statement about the packets temporal history. In this context, SCP defines four different delay values.

Send2Recv: The time passed between transmission of a packet by the SimCon target instance and its reception by the SimCon home instance.

$$\boxed{\Delta t_{s2r} = t^H_{recv0} - t^H_{trm0} = t^H_{recv0} - (t^{T \to H} + t^T_{trm0})} \tag{2.7}$$

t^H_{recv0} is equivalent to the RECV0 header field and t^T_{trm0} is equivalent to the TRM0 header field.

Acqu2Output: The time passed between acquisition of a packet's data by the SimCon target instance and its output by the SimCon home instance.

$$\boxed{\Delta t_{a2o} = t^H_{out} - t^H_{acqu0} = t^H_{out} - (t^{T\to H} + t^T_{acqu0})} \tag{2.8}$$

t^T_{acqu0} is equivalent to the ACQU0 header field.

DeadTime in seconds: The time passed between acquisition of a forward packet's data by the SimCon home instance and the associated return packet's output by the SimCon home instance.

$$\boxed{\Delta t_{dead} = t^H_{out} - t^H_{acqu1}} \tag{2.9}$$

t^H_{acqu1} is equivalent to the return packet's ACQU1 header field.

DeadTime in steps: Equivalent to DeadTime in seconds except simulation steps are considered instead of continuous simulation time.

$$\boxed{\Delta n_{dead} = n^H_{out} - n^H_{acqu1}} \tag{2.10}$$

n^H_{acqu1} is equivalent to the return packet's ACQU1_TIMESTEP header field.

Fig.2.8 illustrates these different delay values. Note that DeadTime in seconds and DeadTime in

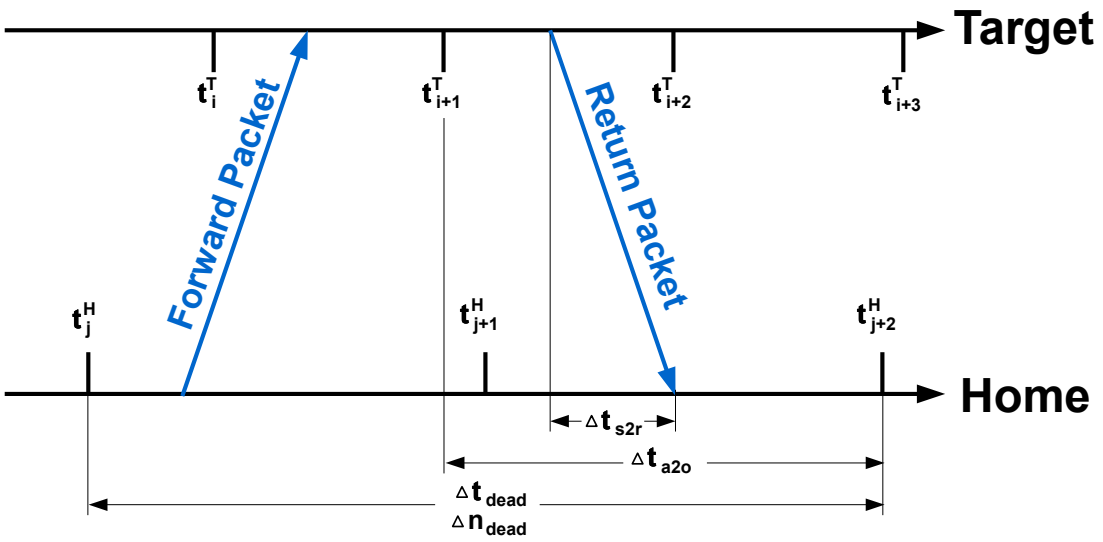

Figure 2.8.: Calculation of Delay Values

steps does not depend on the time transformation. Moreover, DeadTime in steps doesn't depend on any continuous time index. The simulation steps are logged by SimCon itself. The latter delay value is exact by definition. The accuracy of all other delay values depends at least on the accuracy of the provided time index or even on the time transformation, which relies on the delay packet mechanism.

The concept of determining a reference time using the delay packet mechanism is used for monitoring only. Depending on other traffic in the network, it may not be reliable or imprecise. However, as outlined in Sec.2.3, SCP is to provide reliable information about correlation between home and target simulation time. Since both simulations run with a fixed step size, the time step can be used to measure passed simulation time. Assuming that the sample times of target and home simulation are known, a reference home time step with associated reference target timestep defines the correlation unambiguously.

Given some point in time in the form of a timestep, expressed in home time frame n^H. Then its representation expressed in target time frame yields

$$\boxed{n^T = \left(n^H - n^H_{ref}\right) \cdot \frac{\Delta t_{sample,h}}{\Delta t_{sample,t}} + n^T_{ref}} \tag{2.11}$$

with the sample time of the target simulation $\Delta t_{sample,t}$ and of the home simulation $\Delta t_{sample,h}$.

The reference time steps are calculated using the `PACKET_TYPE_REF` packet which has to be sent at each time step. Its ACQU0_TIMESTEP field has to contain the time step when the packet is sent. Suppose such a packet is received by a SimCon home instance at time step n^H. Then the reference time steps yield

$$n^T_{ref} = \text{ACQU0\_TIMESTEP} \tag{2.12}$$

$$n^H_{ref} = n^H + 1 \tag{2.13}$$

The reason for $+1$ in (2.13) readily becomes clear when considering the example depicted in Fig.2.9. The target simulation (i.e. EPOS ACS/RT) is running at a considerably larger sample frequency than

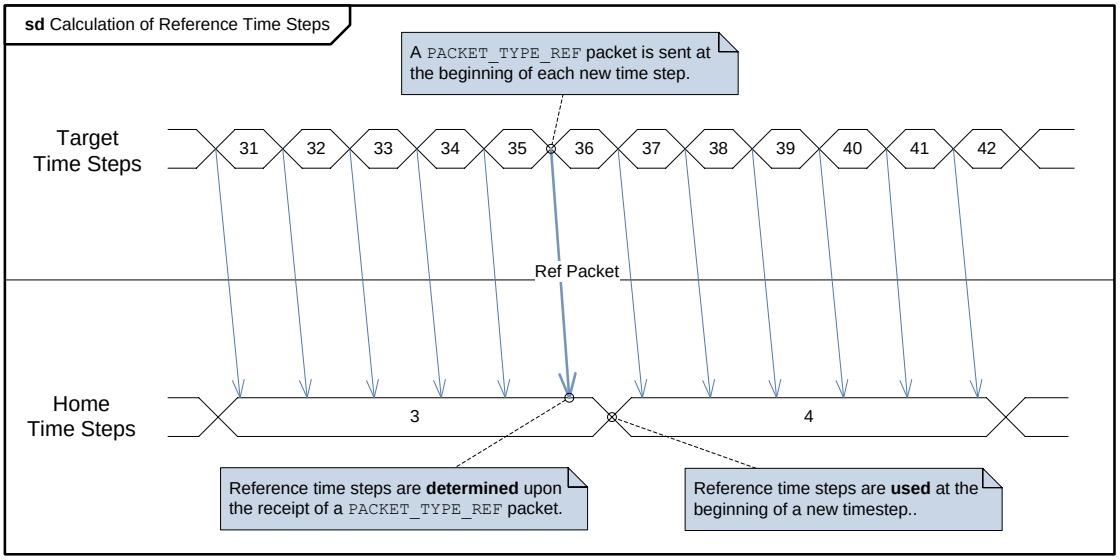

Figure 2.9.: UML Timining Diagram Illustrating Calculation of Reference Timesteps

the home simulation (FF-Testbed FCC). For now, only `PACKET_TYPE_REF` packets sent by the target simulation are considered. The transmission is carried out approximately at the beginning of each target time step. Consider the packet sent at the beginning of time step 36, hence

$$\text{ACQU0\_TIMESTEP} = 36$$

It is received during time step 3 expressed in home time frame, hence

$$n^H = 3$$

Now calculate reference time steps *w*ithout $+1$.

$$n^T_{ref} = \text{ACQU0\_TIMESTEP} = 36$$
$$n^H_{ref} = n^H = 3$$

In general, the reference timesteps are used by the home simulation at the beginning of each time step (and not in-between). This is due to the fact that actions in a Simulink simulation make sense

only at discrete points in time when inputs are read and outputs are calculated. The results calculated in-between, like the reference time steps (in additional threads), can be communicated to the rest of the simulation only at these discrete points in time. In the example, the calculated values are used at the beginning of timestep 4 by the home simulation. This means according to (2.11) that when the home simulation executes time step 3, e.g. at the beginning of time step 3, the target simulation should be at time step 36 which is wrong, of course. This is due to the fact that the last dicisive calculation of reference time steps took place shortly before the switch from 3 to 4. Adding 1 resolves this problem, hence $+1$ in (2.13).

For reference time steps calculated at the target simulation running at the higher frequency this conjuncture has a neglectable impact. However, since SCP doesn't make a difference between a home or a target simulation (pure definition of point of view), e.g. all instances are equal, (2.13) is applied non the less.

2.6. Remote Simulation Protocol (RSP)

As already illustrated in Sec.2.4, RSP includes a client and a server. The client is embedded in the remote simulation. The server is part of the simulation on the ACS/RT. Both communicate via RSP packets which are transported by SCP as data sections of SCP packets. As a basis, RSP structure is described in Sec.2.6.1. Before presenting any details, the basic interaction between server and client is given, along with required definitions and expressens, in Sec.2.6.2.Then, in Sec.2.6.3 client behaviour is presented and in Sec.2.6.4 server behaviour. Finally, it is shown how both interact in detail in Sec.2.6.5, thus realizing a remote simulation.

2.6.1. RemoteSim Packet Structure

A RSP packet is essentially a vector of doubles. The fields have different meanings, depending on usage and whether the client or the server sends a packet. Fig.2.10 depicts packet structure. The numbers at the left indicate the index of the value in the packet. Fields shaded blue have the same or at least similar meaning in all cases. Fields shaded red may differ in meaning as indicated by the vertical division. This means that Chaser Position X and Chaser Force X are represented by the same field, but in some cases the meaning corresponds to the former and in some cases to the latter. Subsequently, the packet fields are treated separately for packets sent by the RSP client and for packets sent by the RSP server.

2.6.1.1. RSP Packet sent by Client/received by Server

For packets sent by the client, the meaning of the different fields depends strongly on the purpose or type of the packet, specified by the field CMD. Therefore, the description is arranged according to the specific value of CMD.

CMD = `CMD_NOT_A_CMD`: Packets with this CMD value are pure dummy packets. They are sent as long as the client waits for a valid connection or for the server to get ready. All other fields have no meaning.

CMD = `CMD_INIT`: This is the packet containing all the information to realize starting conditions, set up and begin a proper simulation.

- **Timestep**: This is the time step expressed in server time frame, when the robots shall have reached the initial conditions and the simulation can start seamlessly.
- **Chaser State**: The initial state of the chaser spacecraft, e.g. the chaser robot.
- **Target State**: The initial state of the target spacecraft, e.g. the target robot.
- **Chaser Force/Torque**: No meaning.
- **Target Force/Torque**: No meaning.
- **CoSy**: The type of coordinate transformation to be used for the simulation. Valid values are: `COSY_FEEDTHROUGH`, `COSY_ECI_2_CLW`
- **Mode**: The type of commanding to be used for the simulation. Valid values are: `MODE_TRAJECTORY`, `MODE_FORCE_TORQUE`

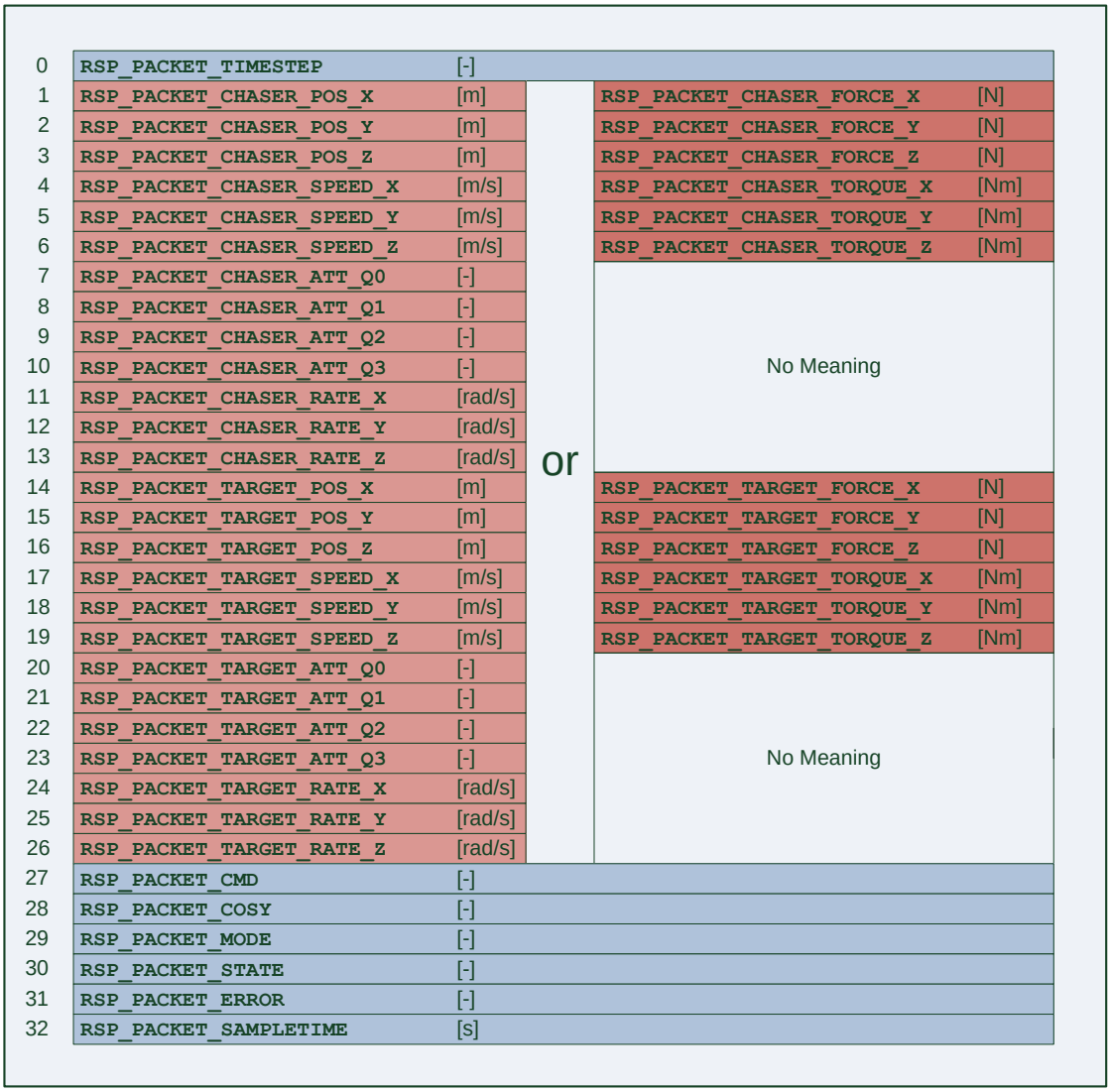

Figure 2.10.: RemoteSim Packet Structure

- **State**: Current client state. See Sec.2.6.3.
- **Error**: No meaning.
- **Sampletime**: The sample time of the remote simulation. The server uses this value to calculate timing for the simulation.

CMD = `CMD_TRAJECTORY`: This packet is sent regularly to directly command the robots' states. The simulation must have been initialized with Mode = `MODE_TRAJECTORY`.

- **Timestep**: This is the time step expressed in server time frame chaser state and target state correspond to. The time step in this packet must match the next command time step calculated by the RemoteSim server.
- **Chaser State**: The state of the chaser spacecraft, e.g. the chaser robot, to be commanded.
- **Target State**: The state of the target spacecraft, e.g. the target robot, to be commanded.
- **Chaser Force/Torque**: No meaning.
- **Target Force/Torque**: No meaning.
- **CoSy**: The type of coordinate transformation as initilized in the `CMD_INIT` packet.
- **Mode**: The type of commanding as initilized in the `CMD_INIT` packet.
- **State**: Current client state. See Sec.2.6.3.

- **Error**: No meaning.
- **Sampletime**: No meaning.

CMD = `CMD_FORCE_TORQUE`: This packet is sent regularly to command force and torque which is fed to the integrator embedded in the ACS/RT simulation. The simulation must have been initialized with Mode = `MODE_FORCE_TRAJECTORY`.

- **Timestep**: This is the time step expressed in server time frame chaser force/torque and target force/torque correspond to. The time step in this packet must match the next command time step calculated by the RemoteSim server.
- **Chaser State**: No meaning.
- **Target State**: No meaning.
- **Chaser Force/Torque**: Force and torque for the integrator of the chaser spacecraft, e.g. chaser robot, to be commanded.
- **Target Force/Torque**: Force and torque for the integrator of the target spacecraft, e.g. target robot, to be commanded.
- **CoSy**: The type of coordinate transformation as initilized in the `CMD_INIT` packet.
- **Mode**: The type of commanding as initilized in the `CMD_INIT` packet.
- **State**: Current client state. See Sec.2.6.3.
- **Error**: No meaning.
- **Sampletime**: No meaning.

2.6.1.2. RSP Packet sent by Server/received by Client

Here, no distinction has to be made concerning the field CMD. The meaning of the different values is the same in all stages

- **Timestep**: No meaning.
- **Chaser State**: Current state of the chaser spacecraft, e.g. the chaser robot.
- **Target State**: Current state of the target spacecraft, e.g. the target robot.
- **Chaser Force/Torque**: No meaning.
- **Target Force/Torque**: No meaning.
- **CMD**: No meaning.
- **CoSy**: The type of coordinate transformation as initilized in the `CMD_INIT` packet. If not yet initialized: `COSY_NOT_A_COSY`
- **Mode**: The type of commanding as initilized in the `CMD_INIT` packet. If not yet initialized: `MODE_NOT_A_MODE`
- **State**: Current server state. See Sec.2.6.4.
- **Error**: Current server error.
- **Sampletime**: The sample time of the EPOS ACS/RT simulation. The client uses this value to calculate timing for the simulation.

2.6.1.3. RSP and SCP

SCP is used to transmit and recieve RSP packets. A RSP packet is thus the data part of a SCP packet, it is encapsulated in the SCP packet compliant with the network layer concept (see Sec.2.2). This is indicated in Fig.2.11.

2.6.2. Basic Client-Server Interaction

Before discussing the behaviour of client and server in detail, the basic interaction is explained in this section. The principle simulation process is presented and some definitions are made as a preparation for subsequent sections. Realization of starting conditions, seamless transition to the simulation and some simulation steps are depicted in the UML timing diagram Fig.2.12 exemplarily. It shows the timeline of a RemoteSim-Client and of a RemoteSim-Server, the numbers indicating

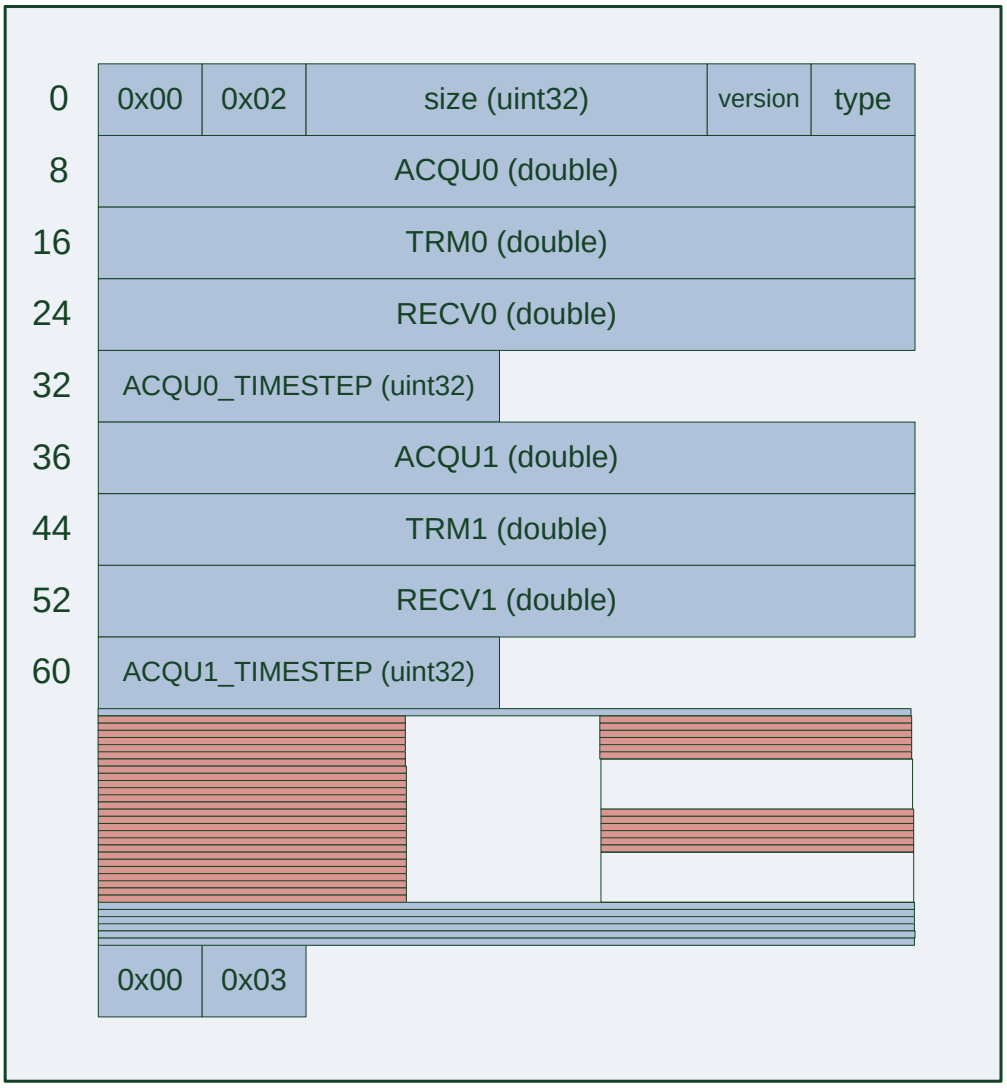

Figure 2.11.: RemoteSim Packet as Data Section in SimCon Packet

time steps. Moreover, RSP packets sent by the client and received by the server are shown as blue arrows. Another timeline in the RemoteSim-Server partition shows the action taken during a certain time period. The various annotations will be explained step by step.

In general, as already outlined in Sec.2.3, the remote simulation along with the RemoteSim-Client runs at a considerably lower sample frequency than the EPOS ACS/RT simulation, e.g. the RemoteSim-Server. This is also indicated in Fig.2.12. In this example 1 client step corresponds to 6 server steps. This relation can be expressed as a ratio, hereby defined server client step ratio

$$\rho_{sample} = \text{round}\left(\frac{\Delta t_{sample,c}}{\Delta t_{sample,s}}\right) = \text{round}\left(\frac{\text{number of server steps per sec}}{\text{number of client steps per sec}}\right) \tag{2.14}$$

with sample time of client $\Delta t_{sample,c}$ and sample time of server $\Delta t_{sample,s}$. In the example

$$\rho_{sample} = 6$$

RSP assumes that the server client step ratio is an integer, e.g. that the number of server steps per second is an integer multiple of the number of client steps per second.

Fig.2.12 assumes that facility initialization has already been carried out successfully and that a valid network connection has been established. At some point, the client sends a `CMD_INIT` packet which contains not only the robots' initial states but also the time step these states have to be reached. Here

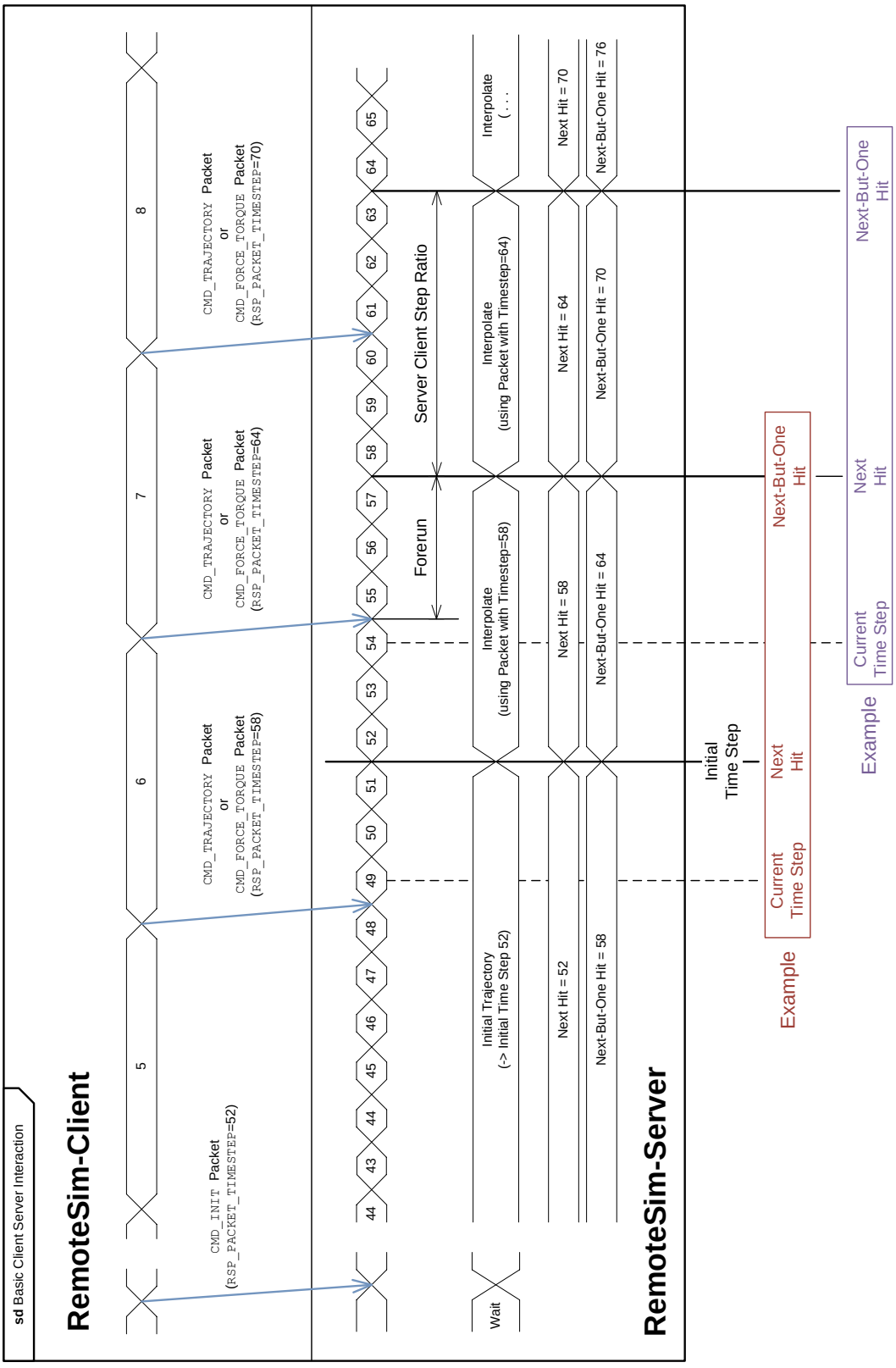

Figure 2.12.: UML Timing Diagram of Basic Client-Server Interaction

the inital time step is 52. The server receives this CMD, calculates an initial trajectory (indicated by the lower lifeline) which ends at the specified initial timestep. Shortly before this time step is due, the client sends the first `CMD_TRAJECTORY` packet (or `CMD_FORCE_TORQUE` for commanding forces and torque). Its field `RSP_PACKET_TIMESTEP` is 58 and means that the transmitted states have to be realized at this time step. As soon as the inital time step is reached, the server calculates an interpolated trajectory, ranging from the current state to the state specified in the `CMD_TRAJECTORY` packet at time step 58. During this period, the client sends the next `CMD_TRAJECTORY` packet, now with `RSP_PACKET_TIMESTEP` 64. As soon as time step 58 is reached, the server again calculates an interpolated trajectory, now ranging to time step 64. And so forth. In this way, client and server are interconnected.

Given some server time step, the closest future time step when an interpolation is calculated based on a received CMD packet is denoted next hit. The time step representing the end-point of the aforementioned interpolation is denoted next-but-one hit. Fig.2.12 shows two examples. The first is highlighted red. Given a current time step 49, the next hit is 52 (here the initial time step) and the next-but-one hit is 58. The second is highlighted purple. Given a current time step of 54, the next hit is 58 and the next-but-one hit is 64. In this context, a received CMD packet's `RSP_PACKET_TIMESTEP` field must match the next-but-one hit with respect to the time step of receipt. As soon as the next hit is reached, the former next-but-one hit becomes the next hit and the new next-but-one hit is the former next hit with the server client step ratio added. This is indicated by the two lowest lifelines in the server partition. Note that this happens only if a proper CMD packet has been received for the next interpolation interval. Denote next hit

$$n_{\text{hit}}(n) \tag{2.15}$$

and next-but-one Hit

$$n_{\text{hit}+1}(n) \tag{2.16}$$

both as functions of a time step. Then the above given examples can be written as

$$n_{\text{hit}}(49) = 52$$
$$n_{\text{hit}+1}(49) = 58$$

and

$$n_{\text{hit}}(54) = 58$$
$$n_{\text{hit}+1}(54) = 64$$

The number of steps between next hit and next-but-one hit equals the server client step ratio.

$$n_{\text{hit}+1}(n) - n_{\text{hit}}(n) = \rho_{\text{sample}} \tag{2.17}$$

The client determines the inital time step expressed in server time frame on the basis of a parameter which specifies the time span to be exploited for the initial trajectory. The server knows its own sample time and the client's sample time which has been transmitted in the `CMD_INIT` packet. This information can be used to calculate the server client step ratio and thus the hits. A CMD packets `RSP_PACKET_TIMESTEP` field must match the calculated hits. Otherwise, this would indicate that something is wrong with timing and the configuration is flawed.

The latest (reasonable) point in time at which a CMD packet can be received regularly is the next hit. At this time step, the next interpolated trajectory is calculated which requires the next CMD packet. Example: The packet with time step 64 is received at time step 55. The latest possible step for reciept would be 57 or (mathematically) precisely at the beginning of step 58. Since ethernet is non-deterministic, it may sometimes take longer to transmit a packet and sometimes it may take less time. Therefore, the client determines the initial step such that there are a few steps between transmission of a CMD packet and the next hit at the server. A measure for the time between the next hit and the receipt of the (right) CMD packet n_{recv} is subsequently denoted forerun ν. It is expressed with respect to the server client step ratio

$$\nu = \frac{n_{\text{hit}}(n_{\text{recv}}) - n_{\text{recv}}}{\rho_{\text{sample}}} \tag{2.18}$$

Ideally it is independent of time, e.g. there is no temporal drift between remote simulation and ACS/RT simulation. It is a parameter of RSP. In the example in Fig.2.12 forerun yields

$$\nu = \frac{58 - 55}{6} = 0.5$$

RSP can catch the error of a late CMD packet, e.g. a CMD packet that is received after next hit. Next hit and next-but-one hit do not change (switch to the next period) until the required packet is received delayed. Then interpolation is calculated and the mechanism goes on as usual. This process will be explained in Secs.2.6.4 and 2.6.5 in detail.

One note about packet transmissions: Both client and server send a RSP packet at each time step. If in this documentation a packet is said to be sent, then the packet's fields are set to the according values. This packet is sent at each time step until its fields are changed. However, RSP is designed in that way. Therefore it only considers the change in received packet field values, it seems as if a packet would be sent only once and the statement "a packet is sent" or "a packet is received" is totally true from a functional point of few.

2.6.3. RemoteSim-Client Behaviour

2.6.3.1. RSP Client Parameters, Inputs and Outputs

The client part of RSP requires several parameters for operation. Tab.2.4 gives an overview with name, symbol and a short description.

Table 2.4.: Parameters of RSP Client

RSP_CLI_PARAM_MODE	-	Type of commanding. Valid values: MODE_TRAJECTORY or MODE_FORCE_TORQUE. Mode is transmitted to the server with the CMD_INIT packet.
RSP_CLI_PARAM_COSY	-	Type of coordinate transformation. Valid values: COSY_FEEDTHROUGH or COSY_ECI_2_CLW. CoSy is transmitted to the server with the CMD_INIT packet.
RSP_CLI_PARAM_FORERUN	ν	Desired forerun. The forerun is used in calculating the initial time step in server time frame. COSY_FEEDTHROUGH or COSY_ECI_2_CLW.
RSP_CLI_PARAM_INIT_TIMESPAN	Δt_{init}	Timespan for the robots to reach the inital conditions beginning from receipt of the CMD_INIT packet. It is used in calculating the initial time step in server time frame.
RSP_CLI_PARAM_SAMPLETIME	$\Delta t_{sample,c}$	Sample time of the remote simulation in seconds.

The client part of RSP has several inputs updated at each time step. Tab.2.5 gives an overview with name, symbol and a short description.

Table 2.5.: Inputs of RSP Client

RSP_CLI_IN_CHASER_REQ_POS RSP_CLI_IN_CHASER_REQ_SPEED RSP_CLI_IN_CHASER_REQ_ATT RSP_CLI_IN_CHASER_REQ_RATE	$\boldsymbol{r}_{req,c}^{U}$ $\dot{\boldsymbol{r}}_{req,c}^{U}$ $\left(q_{req,c}\right)^{U}$ $\left(\boldsymbol{\omega}_{req,c}^{Body}\right)^{U}$	Requested chaser position in $[m]$, speed in $[m/s]$, attitude as quaternion and angular speed in $[rad/s]$, calculated by the remote simulation.
RSP_CLI_IN_TARGET_REQ_POS RSP_CLI_IN_TARGET_REQ_SPEED RSP_CLI_IN_TARGET_REQ_ATT RSP_CLI_IN_TARGET_REQ_RATE	$\boldsymbol{r}_{req,t}^{U}$ $\dot{\boldsymbol{r}}_{req,t}^{U}$ $\left(q_{req,t}\right)^{U}$ $\left(\boldsymbol{\omega}_{req,t}^{Body}\right)^{U}$	Requested target position in $[m]$, speed in $[m/s]$, attitude as quaternion and angular speed in $[rad/s]$, calculated by the remote simulation.
RSP_CLI_IN_CHASER_REQ_FORCE RSP_CLI_IN_CHASER_REQ_TORQUE	$\boldsymbol{F}_{req,c}$ $\boldsymbol{T}_{req,c}$	The requested force in $[N]$ and torque in $[Nm]$ for the chaser to be fed to the integrator at the ACS/RT if operating in MODE_FORCE_TORQUE. It is calculated by the user simulation.
RSP_CLI_IN_TARGET_REQ_FORCE RSP_CLI_IN_TARGET_REQ_TORQUE	$\boldsymbol{F}_{req,t}$ $\boldsymbol{T}_{req,t}$	The requested force in $[N]$ and torque in $[Nm]$ for the target to be fed to the integrator at the ACS/RT if operating in MODE_FORCE_TORQUE. It is calculated by the user simulation.
RSP_CLI_IN_CLIENT_REF_TIMESTEP	n_{ref}^{C}	The client reference time step is supplied by the SCP layer. Together with SERVER_REF_TIMESTEP, it provides information about the correlation between client time steps and server time steps.
RSP_CLI_IN_SERVER_REF_TIMESTEP	n_{ref}^{S}	The server reference time step is supplied by the SCP layer. Together with CLIENT_REF_TIMESTEP, it provides information about the correlation between client time steps and server time steps.
RSP_CLI_IN_CONNECTION_OK	-	This signal is coming from the SCP layer and indicates whether a connection to a RemoteSim-Server is intact (1) or not (0).
RSP_CLI_IN_RECVD_RSP_PACKET	-	The RSP packet coming from the RemoteSim-Server, provided by SCP.

The client part of RSP has several outputs updated at each time step. Tab.2.6 gives an overview with name, symbol and a short description.

Table 2.6.: Outputs of RSP Client

RSP_CLI_OUT_CHASER_CURR_POS RSP_CLI_OUT_CHASER_CURR_SPEED RSP_CLI_OUT_CHASER_CURR_ATT RSP_CLI_OUT_CHASER_CURR_RATE	$\boldsymbol{r}_{curr,c}^{U}$ $\dot{\boldsymbol{r}}_{curr,c}^{U}$ $(q_{curr,c})^{U}$ $\left(\boldsymbol{\omega}_{curr,c}^{Body}\right)^{U}$	Current position in $[m]$, current speed in $[m/s]$, attitude as a quaternion and angular speed in $[rad/s]$ of the chaser, received from RemoteSim-Server.
RSP_CLI_OUT_TARGET_CURR_POS RSP_CLI_OUT_TARGET_CURR_SPEED RSP_CLI_OUT_TARGET_CURR_ATT RSP_CLI_OUT_TARGET_CURR_RATE	$\boldsymbol{r}_{curr,t}^{U}$ $\dot{\boldsymbol{r}}_{curr,t}^{U}$ $(q_{curr,t})^{U}$ $\left(\boldsymbol{\omega}_{curr,t}^{Body}\right)^{U}$	Current position in $[m]$, current speed in $[m/s]$, attitude as a quaternion and angular speed in $[rad/s]$ of the target, received from RemoteSim-Server.
RSP_CLI_OUT_SERVER_STATE	-	Current state of the RemoteSim-Server.
RSP_CLI_OUT_SERVER_ERROR	-	Current error of the RemoteSim-Server.
RSP_CLI_OUT_STATE	-	Current client state.
RSP_CLI_OUT_SIM_ENABLE	-	Simulation enable flag. Signals that initial conditions have been reached and actual simulation can begin.
RSP_CLI_OUT_SEND_RSP_PACKET	-	RSP packet to be sent to RemoteSim-Server.

2.6.3.2. RSP Client State Machine

Fig.2.13 shows an UML state diagram to illustrate RemoteSim-Client behaviour. This state machine is run at each time step of the remote simulation. At the beginning, the client is in STATE_WAIT_FOR_CONNECTION. As the expression suggests, the client rests in this state as long as no RemoteSim-Server is found. As soon as the SCP layer signals a valid connection (SCP_OUT_CONNECTION_OK), the state machine transits to STATE_WAIT_FOR_SERVER_READY_FOR_INIT. Not before the server state changes to STATE_READY_FOR_INIT_CMD does the client transit to STATE_SIM_INIT. This transition is accompanied by two activities. First the timing for the simulation is calculated, then the output packets' fields are set according to CMD_INIT which is logically equivalent with transmitting this packet. Both will be discussed farer below. As soon as the initial time step $n_{init,c}^{C}$ has been reached, the client transits to the next state. Depending on SCP_PARAM_MODE transmitted in the CMD_INIT packet as RSP_PACKET_MODE, this may be STATE_SIM_TRAJECTORY or STATE_SIM_FORCE_TORQUE. In the former, the trajectory is calculated directly and in the latter force and torque are commanded. In any case, with the transition the actual user simulation is enabled such that a seamless transition from initial values to nominal simulation is possible (RSP_CLI_OUT_SIM_ENABLE). Note that the enable signal is activated *one step before* the client reaches STATE_SIM_TRAJECTORY or STATE_SIM_FORCE_TORQUE for the first time, e.g. *one step before* the actual simulation is to start. This implies that a unit delay has to be included in the Simulink model. Such a unit dely avoids an algebraic loop which would inevitably occur otherwise.

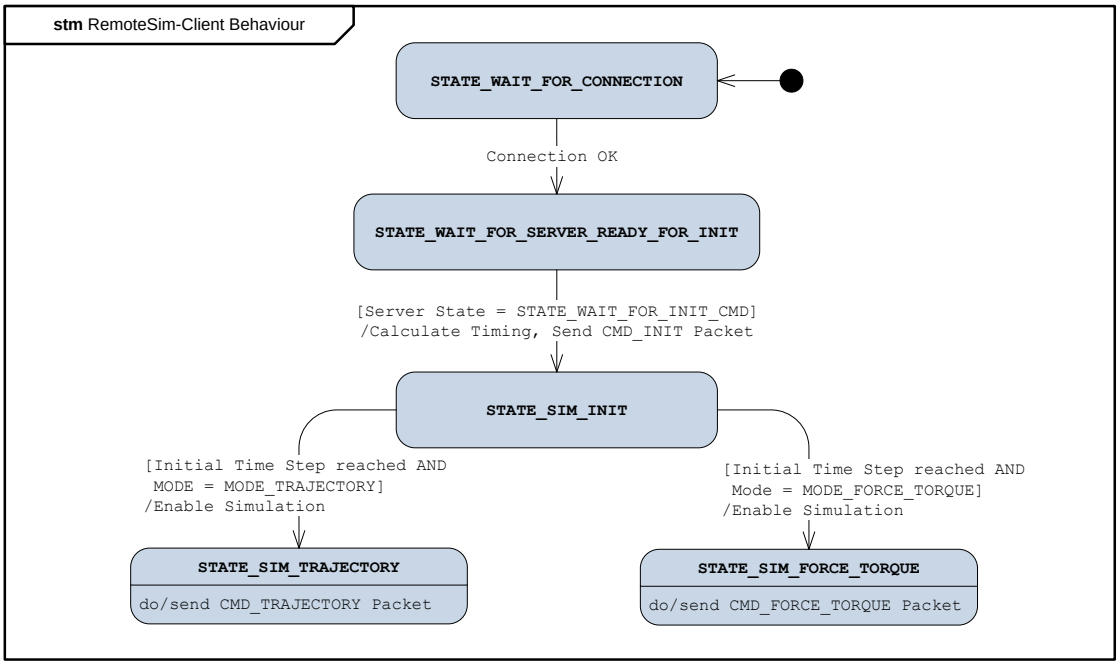

Figure 2.13.: UML state diagram of RemoteSim-Client Behaviour

2.6.3.3. Calculation of Timing

In essence, it is one value which has to be determined to achieve a proper remote simulation/EPOS interrelation: initial time step expressed in server time frame n^S_{init}.

With the initial timespan Δt_{init} and the client sample time $\Delta t_{sample,c}$ given as RSP parameters as well as the current time step n^C_{curr}, the initial time step for the client

$$n^C_{init,c} = \text{round}\left(\frac{\Delta t_{init}}{\Delta t_{sample,c}}\right) + n^C_{curr} \tag{2.19}$$

$n^C_{init,c}$ is the time step when the client transits to regular simulation state. At $n^C_{init,c}+1$ it sends the first CMD packet associated with the first commanded states after the initial states. This CMD must be received by RemoteSim-Server shortly before the initial time step of the server is reached, the "shortly before" being determined by forerun ν. In the example of Fig.2.12, $n^C_{init,c} = 4$. Thus, the initial time step of RemoteSim-Server expressed in server time frame follows

$$n^S_{init,s} = n^S_{init,c} + \text{round}\left[(1+\nu)\cdot\rho_{sample}\right] \tag{2.20}$$

The client initial time step is needed in server time frame. Using (2.11), it follows

$$n^S_{init,c} = \left(n^C_{init,c} - n^C_{ref}\right)\cdot\rho_{sample} + n^S_{ref} \tag{2.21}$$

2.6.3.4. RSP Client Output Behaviour

Formally, the RSP client is a Mealy State Machine: Its outputs not only depend on the state but also on the inputs. Here, the only relevant input is `RSP_PACKET_STATE` of `RSP_CLI_IN_RECVD_RSP_PACKET` which is associated with current state of RemoteSim-Server.

Tab.2.7 summarizes the fields of `RSP_CLI_OUT_SEND_RSP_PACKET` depending on RemoteSim-Client state and RemoteSim-Server state as input.

In any case, `RSP_PACKET_CMD` must contain a valid value. As long as the client waits for a valid connection or for the server to get ready for a `CMD_INIT` packet, this field equals `CMD_NOT_A_CMD`.

RSP_PACKET_STATE	STATE_WAIT_FOR_CONNECTION	STATE_WAIT_FOR_SERVER_READY_FOR_INIT Server State≠STATE_WAIT_FOR_INIT_CMD	STATE_WAIT_FOR_SERVER_READY_FOR_INIT Server State=STATE_WAIT_FOR_INIT_CMD	STATE_SIM_INIT	STATE_SIM_TRAJECTORY	STATE_SIM_FORCE_TORQUE
RSP_PACKET_TIMESTEP	N/M	N/M	$n^S_{init,s}$	N/M	$n^S_{init+1,s}$	$n^S_{init+1,s}$
RSP_PACKET_CHASER_POS/_FORCE	N/M	N/M	$\boldsymbol{r}^{\mathcal{U}}_{req,c}$	N/M	$\boldsymbol{r}^{\mathcal{U}}_{req,c}$	$\boldsymbol{F}_{req,c}$
RSP_PACKET_CHASER_SPEED/_TORQUE	N/M	N/M	$\dot{\boldsymbol{r}}^{\mathcal{U}}_{req,c}$	N/M	$\dot{\boldsymbol{r}}^{\mathcal{U}}_{req,c}$	$\boldsymbol{T}_{req,c}$
RSP_PACKET_CHASER_ATT	N/M	N/M	$\left(q_{req,c}\right)^{\mathcal{U}}$	N/M	$\left(q_{req,c}\right)^{\mathcal{U}}$	N/M
RSP_PACKET_CHASER_RATE	N/M	N/M	$\left(\boldsymbol{\omega}^{Body}_{req}\right)^{\mathcal{U}}$	N/M	$\left(\boldsymbol{\omega}^{Body}_{req}\right)^{\mathcal{U}}$	N/M
RSP_PACKET_TARGET_POS/_FORCE	N/M	N/M	$\boldsymbol{r}^{\mathcal{U}}_{req,c}$	N/M	$\boldsymbol{r}^{\mathcal{U}}_{req,c}$	$\boldsymbol{F}_{req,t}$
RSP_PACKET_TARGET_SPEED/_TORQUE	N/M	N/M	$\dot{\boldsymbol{r}}^{\mathcal{U}}_{req,c}$	N/M	$\dot{\boldsymbol{r}}^{\mathcal{U}}_{req,c}$	$\boldsymbol{T}_{req,t}$
RSP_PACKET_TARGET_ATT	N/M	N/M	$\left(q_{req,c}\right)^{\mathcal{U}}$	N/M	$\left(q_{req,c}\right)^{\mathcal{U}}$	N/M
RSP_PACKET_TARGET_RATE	N/M	N/M	$\left(\boldsymbol{\omega}^{Body}_{req}\right)^{\mathcal{U}}$	N/M	$\left(\boldsymbol{\omega}^{Body}_{req}\right)^{\mathcal{U}}$	N/M
RSP_PACKET_CMD	CMD_NOT_A_CMD	CMD_NOT_A_CMD	CMD_INIT	CMD_INIT	CMD_TRAJECTORY	CMD_FORCE_TORQUE
RSP_PACKET_COSY	N/M	N/M	RSP_CLI_PARAM_COSY	N/M	RSP_CLI_PARAM_COSY	RSP_CLI_PARAM_COSY
RSP_PACKET_MODE	N/M	N/M	RSP_CLI_PARAM_MODE	N/M	RSP_CLI_PARAM_MODE	RSP_CLI_PARAM_MODE
RSP_PACKET_ERROR	N/M	N/M	N/M	N/M	N/M	N/M
RSP_PACKET_SAMPLETIME	N/M	N/M	$\Delta t_{sample,c}$	N/M	N/M	N/M

Table 2.7.: Fields of RSP_CLI_OUT_SEND_RSP_PACKET (RSP Client) depending on Client State and Inputs (only Server State). N/M - No Meaning.

RSP_CLI_OUT_STATE	STATE_WAIT_FOR_CONNECTION	STATE_WAIT_FOR_SERVER_READY_FOR_INIT	STATE_SIM_INIT $n_{curr,c} \neq n_{init,c}$	STATE_SIM_INIT $n_{curr,c} = n_{init,c}$	STATE_SIM_TRAJECTORY/ STATE_SIM_FORCE_TORQUE
RSP_CLI_OUT_CHASER_CURR_POS	N/M	N/M	N/M	N/M	RSP_PACKET_CHASER_POS
RSP_CLI_OUT_CHASER_CURR_SPEED	N/M	N/M	N/M	N/M	RSP_PACKET_CHASER_SPEED
RSP_CLI_OUT_CHASER_CURR_ATT	N/M	N/M	N/M	N/M	RSP_PACKET_CHASER_ATT
RSP_CLI_OUT_CHASER_CURR_RATE	N/M	N/M	N/M	N/M	RSP_PACKET_CHASER_RATE
RSP_CLI_OUT_TARGET_CURR_POS	N/M	N/M	N/M	N/M	RSP_PACKET_TARGET_POS
RSP_CLI_OUT_TARGET_CURR_SPEED	N/M	N/M	N/M	N/M	RSP_PACKET_TARGET_SPEED
RSP_CLI_OUT_TARGET_CURR_ATT	N/M	N/M	N/M	N/M	RSP_PACKET_TARGET_ATT
RSP_CLI_OUT_TARGET_CURR_RATE	N/M	N/M	N/M	N/M	RSP_PACKET_TARGET_RATE
RSP_CLI_OUT_SERVER_STATE	RSP_PACKET_STATE	RSP_PACKET_STATE	RSP_PACKET_STATE	RSP_PACKET_STATE	RSP_PACKET_STATE
RSP_CLI_OUT_SERVER_ERROR	RSP_PACKET_ERROR	RSP_PACKET_ERROR	RSP_PACKET_ERROR	RSP_PACKET_ERROR	RSP_PACKET_ERROR
RSP_CLI_OUT_SIM_ENABLE	0	0	0	1	1

Table 2.8.: Output values of RSP Client depending on Client State and Server State. N/M - No Meaning. All RSP packet fields given here correspond to the received packet from the RemoteSim-Server, e.g. RSP_CLI_IN_RECVD_RSP_PACKET.

As soon as the server transits to STATE_WAIT_FOR_INIT_CMD, RSP_PACKET_CMD must be CMD_INIT and also stays during STATE_SIM_INIT. Depending on the mode of commanding, in STATE_SIM_TRAJECTORY the CMD is CMD_TRAJECTORY and in STATE_SIM_FORCE_TORQUE it is CMD_FORCE_TORQUE, respectively.

The field RSP_PACKET_SAMPLETIME has to contain a valid value only when the CMD_INIT packet is sent for the first time (for a specific simulation), which corresponds to STATE_WAIT_FOR_SERVER_READY_FOR_INIT with server state (received from server via RSP packet) STATE_WAIT_FOR_INIT_CMD. In all other cases, RemoteSim-Server doesn't read that field at all.

RSP_PACKET_ERROR has no meaning, since RSP does not include the possibility of an error occuring at the remote simulation side.

RSP_PACKET_COSY and RSP_PACKET_MODE must be set to the specified parameters RSP_CLI_PARAM_COSY and RSP_CLI_PARAM_COSY, when the CMD_INIT packet is sent the first time (e.g. STATE_WAIT_FOR_SERVER_READY_FOR_INIT and server state STATE_WAIT_FOR_INIT_CMD) as well as during commanding with CMD_TRAJECTORY and CMD_FORCE_TORQUE. In all other cases, the fields' values have no impact.

When the CMD_INIT packet is sent for the first time, chaser and target state must equal the initial states for the simulation run. It is required that these inital S/C states are supplied to the RemoteSim-Client's inputs. Later, these inputs will be used to get the S/C states during STATE_SIM_TRAJECTORY. So the same inputs are used for regular simulation commands and initial states. The simulation enable signal (see below) is used to allow the model creator to switch between these values. During STATE_SIM_FORCE_TORQUE, packet fields are used differently. RSP_PACKET_CHASER_FORCE, RSP_PACKET_CHASER_TORQUE, RSP_PACKET_TARGET_FORCE and RSP_PACKET_TARGET_TORQUE are set to force and torque given by the client inputs RSP_CLI_IN_CHASER_FORCE, RSP_CLI_IN_CHASER_TORQUE, RSP_CLI_IN_TARGET_FORCE and RSP_CLI_IN_TARGET_TORQUE.

Tab.2.8 summarizes the RemoteSim-Client outputs (except RSP_CLI_OUT_SEND_RSP_PACKET) depending on client state which formally includes the current time step.

In the process of simulation initialization, RSP_CLI_OUT_SIM_ENABLE changes from 0 to 1, in STATE_SIM_INIT if the initial time step is reached. This is one step ahead of the beginning of the actual simulation. Thus, in the Simulink model a unit delay can be included to avoid an algebraic loop. In states before, the enable signal is 0 and in all states after 1.

The outputs RSP_CLI_OUT_SERVER_STATE and RSP_CLI_OUT_SERVER_ERROR always equal the fields RSP_PACKET_STATE and RSP_PACKET_ERROR of the RSP packet received from RemoteSim-Server.

Finally, the client outputs associated with current chaser and target state equal the fields RSP_PACKET_CHASER_POS, RSP_PACKET_CHASER_SPEED... of the RSP packet received from RemoteSim-Server. This holds during regular simulation, e.g. during STATE_SIM_TRAJECTORY and STATE_SIM_FORCE_TORQUE. In all other states, the outputs have no meaning.

2.6.4. RemoteSim-Server Behaviour

2.6.4.1. RSP Server Parameters, Inputs and Outputs

The server part of RSP requires several parameters. Tab.2.9 gives an overview with name, symbol and a short description.

Table 2.9.: Parameters of RSP Server

RSP_SVR_PARAM_SPEED_LIMIT	$\dot{r}_{lim}$	Translational speed limit in $[m/s]$, angular speed limit in $[m/s]$, translational acceleration limit in $[m/s^2]$ and angular acceleration limit in $[rad/s^2]$ for both robots. The robots' movements are constrained such that the limits are not violated.
RSP_SVR_PARAM_ANGULAR_SPEED_LIMIT	ω_{lim}	
RSP_SVR_PARAM_ACC_LIMIT	$\ddot{r}_{lim}$	
RSP_SVR_PARAM_ANGULAR_ACC_LIMIT	$\dot{\omega}_{lim}$	

Continued on next page

Table 2.9 – *Continued from previous page*

RSP_SVR_PARAM_CHASER_FACILITY_INIT_POS RSP_SVR_PARAM_CHASER_FACILITY_INIT_ATT	$\boldsymbol{r}_{finit,c}^{CLW}$ $\left(q_{finit,c}\right)^{CLW}$	The chasers's position in $[m]$ and attitude as a quaternion for facility initialization.
RSP_SVR_PARAM_TARGET_FACILITY_INIT_POS RSP_SVR_PARAM_TARGET_FACILITY_INIT_ATT	$\boldsymbol{r}_{finit,t}^{CLW}$ $\left(q_{finit,t}\right)^{CLW}$	The target's position in $[m]$ and attitude as a quaternion for facility initialization.
RSP_SVR_PARAM_SAMPLETIME	$\Delta t_{sample,s}$	Sample time of the EPOS ACS/RT simulation in $[s]$, usually 0.004.
RSP_SVR_PARAM_BRAKE_ACC RSP_SVR_PARAM_ANGULAR_BRAKE_ACC	$\ddot{\boldsymbol{r}}_{brake}$ $\dot{\boldsymbol{\omega}}_{brake}$	The translational acceleration in $[m/s^2]$ and the angular acceleration in $[rad/s^2]$ applied for decelerating the robots, when an error occurs or the remote simulation is stopped.

The server part of RSP has several inputs updated at each time step. Tab.2.10 gives an overview with name, symbol and a short description.

Table 2.10.: Inputs of RSP Server

RSP_SVR_IN_CMD_IF_ENABLE	-	Signal coming from the EPOS CMD Interface, indicating that facility initialization has been completed.
RSP_SVR_IN_CHASER_CURR_POS RSP_SVR_IN_CHASER_CURR_ATT	$\boldsymbol{r}_{curr,c}^{CLW}$ $(q_{curr,c})^{CLW}$	Current chaser position in $[m]$ and attitude as a quaternion, returned by the EPOS CMD Interface.
RSP_SVR_IN_TARGET_CURR_POS RSP_SVR_IN_TARGET_CURR_ATT	$\boldsymbol{r}_{curr,t}^{CLW}$ $(q_{curr,t})^{CLW}$	Current target position in $[m]$ and attitude as a quaternion, returned by the EPOS CMD Interface.
RSP_SVR_IN_CHASER_INT_POS RSP_SVR_IN_CHASER_INT_SPEED RSP_SVR_IN_CHASER_INT_ATT RSP_SVR_IN_CHASER_INT_RATE	$\boldsymbol{r}_{int,c}^{U}$ $\dot{\boldsymbol{r}}_{int,c}^{U}$ $(q_{int,c})^{U}$ $\left(\boldsymbol{\omega}_{int,c}^{Body}\right)^{U}$	Commanded chaser position in $[m]$, speed in $[m/s]$, attitude as a quaternion and angular speed in $[rad/s]$, returned by the integrator embedded in the EPOS ACS/RT simulation. Used with MODE_FORCE_TORQUE only.
RSP_SVR_IN_TARGET_INT_POS RSP_SVR_IN_TARGET_INT_SPEED RSP_SVR_IN_TARGET_INT_ATT RSP_SVR_IN_CHASER_INT_RATE	$\boldsymbol{r}_{int,t}^{U}$ $\dot{\boldsymbol{r}}_{int,t}^{U}$ $(q_{int,t})^{U}$ $\left(\boldsymbol{\omega}_{int,t}^{Body}\right)^{U}$	Commanded target position in $[m]$, speed in $[m/s]$, attitude as a quaternion and angular speed in $[rad/s]$, returned by the integrator embedded in the EPOS ACS/RT simulation. Used with MODE_FORCE_TORQUE only.
RSP_SVR_IN_RESET	-	Input used to confirm an error and reset RemoteSim-Server.

Continued on next page

Table 2.10 – *Continued from previous page*

RSP_SVR_IN_CLIENT_REF_TIMESTEP	n_{ref}^{C}	The client reference time step is supplied by the SCP layer. Together with SERVER_REF_TIMESTEP, it provides information about the correlation between client time steps and server time steps.
RSP_SVR_IN_SERVER_REF_TIMESTEP	n_{ref}^{S}	The server reference time step is supplied by the SCP layer. Together with CLIENT_REF_TIMESTEP, it provides information about the correlation between client time steps and server time steps.
RSP_SVR_IN_CONNECTION_OK	-	This signal is coming from the SCP layer and indicates whether a connection to a RemoteSim-Server is intact (1) or not (0).
RSP_SVR_IN_RECVD_RSP_PACKET	-	The RSP packet coming from the RemoteSim-Client, provided by SCP.

The server part of RSP has several outputs updated at each time step. Tab.2.11 gives an overview with name, symbol and a short description.

Table 2.11.: Outputs of RSP Server

RSP_SVR_OUT_STATE	-	Current server state.
RSP_SVR_OUT_ERROR	-	Current server error.
RSP_SVR_OUT_MODE	-	Type of commanding as initialized by remote simulation, MODE_TRAJECTORY or MODE_FORCE_TORQUE. Before initialization MODE_NOT_A_MODE.
RSP_SVR_OUT_COSY	-	Server type of coordinate transformation as initialized by remote simulation, COSY_FEEDTHROUGH or COSY_ECI_2_CLW. Before initialization COSY_NOT_A_COSY.
RSP_SVR_OUT_CLIENT_STATE	-	Current state of RemoteSim-Client.
RSP_SVR_OUT_CHASER_CMD_POS RSP_SVR_OUT_CHASER_CMD_ATT	$\boldsymbol{r}_{cmd,c}^{CLW}$ $(q_{cmd,c})^{CLW}$	Commanded chaser position in $[m]$ and attitude as a quaternion, to be supplied to the EPOS CMD IF.
RSP_SVR_OUT_TARGET_CMD_POS RSP_SVR_OUT_TARGET_CMD_ATT	$\boldsymbol{r}_{cmd,t}^{CLW}$ $(q_{cmd,t})^{CLW}$	Commanded target position in $[m]$ and attitude as a quaternion, to be supplied to the EPOS CMD IF.

Continued on next page

Table 2.11 – *Continued from previous page*

RSP_SVR_OUT_CHASER_CURR_POS_CLW RSP_SVR_OUT_CHASER_CURR_SPEED_CLW RSP_SVR_OUT_CHASER_CURR_ATT_CLW RSP_SVR_OUT_CHASER_CURR_RATE_CLW	$\boldsymbol{r}_{curr,c}^{CLW}$ $\dot{\boldsymbol{r}}_{curr,c}^{CLW}$ $(q_{curr,c})^{CLW}$ $\left(\boldsymbol{\omega}_{curr,c}^{Body}\right)^{CLW}$	Current chaser position in $[m]$, speed in $[m/s]$, attitude as a quaternion and angular speed in $[rad/s]$, expressed in CLW coordinates.
RSP_SVR_OUT_TARGET_CURR_POS_CLW RSP_SVR_OUT_TARGET_CURR_SPEED_CLW RSP_SVR_OUT_TARGET_CURR_ATT_CLW RSP_SVR_OUT_TARGET_CURR_RATE_CLW	$\boldsymbol{r}_{curr,t}^{CLW}$ $\dot{\boldsymbol{r}}_{curr,t}^{CLW}$ $(q_{curr,t})^{CLW}$ $\left(\boldsymbol{\omega}_{curr,t}^{Body}\right)^{CLW}$	Current target position in $[m]$, speed in $[m/s]$, attitude as a quaternion and angular speed in $[rad/s]$, expressed in CLW coordinates.
RSP_SVR_OUT_CHASER_CURR_POS_U RSP_SVR_OUT_CHASER_CURR_SPEED_U RSP_SVR_OUT_CHASER_CURR_ATT_U RSP_SVR_OUT_CHASER_CURR_RATE_U	$\boldsymbol{r}_{curr,c}^{U}$ $\dot{\boldsymbol{r}}_{curr,c}^{U}$ $(q_{curr,c})^{U}$ $\left(\boldsymbol{\omega}_{curr,c}^{Body}\right)^{U}$	Current chaser position in $[m]$, speed in $[m/s]$, attitude as a quaternion and angular speed in $[rad/s]$, expressed in U (depending on RSP_CLI_PARAM_COSY) coordinates.
RSP_SVR_OUT_TARGET_CURR_POS_U RSP_SVR_OUT_TARGET_CURR_SPEED_U RSP_SVR_OUT_TARGET_CURR_ATT_U RSP_SVR_OUT_TARGET_CURR_RATE_U	$\boldsymbol{r}_{curr,t}^{U}$ $\dot{\boldsymbol{r}}_{curr,t}^{U}$ $(q_{curr,t})^{U}$ $\left(\boldsymbol{\omega}_{curr,t}^{Body}\right)^{U}$	Current target position in $[m]$, speed in $[m/s]$, attitude as a quaternion and angular speed in $[rad/s]$, expressed in U (depending on RSP_CLI_PARAM_COSY) coordinates.
RSP_SVR_OUT_CHASER_REQ_POS_CLW RSP_SVR_OUT_CHASER_REQ_SPEED_CLW RSP_SVR_OUT_CHASER_REQ_ATT_CLW RSP_SVR_OUT_CHASER_REQ_RATE_CLW	$\boldsymbol{r}_{req,c}^{CLW}$ $\dot{\boldsymbol{r}}_{req,c}^{CLW}$ $(q_{req,c})^{CLW}$ $\left(\boldsymbol{\omega}_{req,c}^{Body}\right)^{CLW}$	Requested chaser position in $[m]$, speed in $[m/s]$, attitude as a quaternion and angular speed in $[rad/s]$, expressed in CLW coordinates, received from remote simulation.
RSP_SVR_OUT_TARGET_REQ_POS_CLW RSP_SVR_OUT_TARGET_REQ_SPEED_CLW RSP_SVR_OUT_TARGET_REQ_ATT_CLW RSP_SVR_OUT_CHASER_REQ_RATE_CLW	$\boldsymbol{r}_{req,t}^{CLW}$ $\dot{\boldsymbol{r}}_{req,t}^{CLW}$ $(q_{req,t})^{CLW}$ $\left(\boldsymbol{\omega}_{req,t}^{Body}\right)^{CLW}$	Requested target position in $[m]$, speed in $[m/s]$, attitude as a quaternion and angular speed in $[rad/s]$, expressed in CLW coordinates, received from remote simulation.

Continued on next page

Table 2.11 – *Continued from previous page*

RSP_SVR_OUT_CHASER_REQ_POS_U RSP_SVR_OUT_CHASER_REQ_SPEED_U RSP_SVR_OUT_CHASER_REQ_ATT_U RSP_SVR_OUT_CHASER_REQ_RATE_U	$\boldsymbol{r}_{req,c}^{U}$ $\dot{\boldsymbol{r}}_{req,c}^{U}$ $(q_{req,c})^{U}$ $\left(\boldsymbol{\omega}_{req,c}^{Body}\right)^{U}$	Requested chaser position in $[m]$, speed in $[m/s]$, attitude as a quaternion and angular speed in $[rad/s]$, expressed in U (depending on RSP_CLI_PARAM_COSY) coordinates, received from remote simulation.
RSP_SVR_OUT_TARGET_REQ_POS_U RSP_SVR_OUT_TARGET_REQ_SPEED_U RSP_SVR_OUT_TARGET_REQ_ATT_U RSP_SVR_OUT_CHASER_REQ_RATE_U	$\boldsymbol{r}_{req,t}^{U}$ $\dot{\boldsymbol{r}}_{req,t}^{U}$ $(q_{req,t})^{U}$ $\left(\boldsymbol{\omega}_{req,t}^{Body}\right)^{U}$	Requested target position in $[m]$, speed in $[m/s]$, attitude as a quaternion and angular speed in $[rad/s]$, expressed in U (depending on RSP_CLI_PARAM_COSY) coordinates, received from remote simulation.
RSP_SVR_OUT_CHASER_INT_INIT_POS_U RSP_SVR_OUT_CHASER_INT_INIT_SPEED_U RSP_SVR_OUT_CHASER_INT_INIT_ATT_U RSP_SVR_OUT_CHASER_INT_INIT_RATE_U	$\boldsymbol{r}_{iinit,c}^{U}$ $\dot{\boldsymbol{r}}_{iinit,c}^{U}$ $(q_{iinit,c})^{U}$ $\left(\boldsymbol{\omega}_{iinit,c}^{Body}\right)^{U}$	Initial integrator chaser position in $[m]$, speed in $[m/s]$, attitude as a quaternion and angular speed in $[rad/s]$, expressed in U (depending on RSP_CLI_PARAM_COSY) coordinates, received from remote simulation with CMD_INIT.
RSP_SVR_OUT_TARGET_INT_INIT_POS_U RSP_SVR_OUT_TARGET_INT_INIT_SPEED_U RSP_SVR_OUT_TARGET_INT_INIT_ATT_U RSP_SVR_OUT_CHASER_INT_INIT_RATE_U	$\boldsymbol{r}_{iinit,t}^{U}$ $\dot{\boldsymbol{r}}_{iinit,t}^{U}$ $(q_{iinit,t})^{U}$ $\left(\boldsymbol{\omega}_{iinit,t}^{Body}\right)^{U}$	Initial integrator target position in $[m]$, speed in $[m/s]$, attitude as a quaternion and angular speed in $[rad/s]$, expressed in U (depending on RSP_CLI_PARAM_COSY) coordinates, received from remote simulation with CMD_INIT.
RSP_SVR_OUT_CHASER_INT_FORCE RSP_SVR_OUT_CHASER_INT_TORQUE	$\boldsymbol{F}_{int,c}$ $\boldsymbol{T}_{int,c}$	Commanded force in $[N]$ and torque in $[Nm]$ for the chaser, to be supplied to the integrator.
RSP_SVR_OUT_TARGET_INT_FORCE RSP_SVR_OUT_TARGET_INT_TORQUE	$\boldsymbol{F}_{int,t}$ $\boldsymbol{T}_{int,t}$	Commanded force in $[N]$ and torque in $[Nm]$ for the target, to be supplied to the integrator.
RSP_SVR_OUT_INT_ENABLE	-	Signal for the integrator in EPOS AC-S/RT Simulink model indicating that actual simulation starts.
RSP_SVR_OUT_SEND_RSP_PACKET	-	RSP packet to be sent to RemoteSim-Client.
RSP_SVR_OUT_ACTUAL_FORERUN	v_{actual}	Actual forerun of the most recently received CMD packet.
RSP_SVR_OUT_CHASER_VIOLATION	-	Signal indicating a violation of speed and/or acceleration limits (see Tab.2.9) by the chaser.

Continued on next page

Table 2.11 – *Continued from previous page*

RSP_SVR_OUT_TARGET_VIOLATION	-	Signal indicating a violation of speed and/or acceleration limits (see Tab.2.9) by the target.

2.6.4.2. RSP Server State Machine

Similar to RemoteSim-Client, server behaviour can be described by a state machine. Fig.2.14 shows the UML state diagram. The state machine can be considered to be run at each time step. Hence, transitions can occur at each time step.

The RSP server starts in STATE_FACILITY_INIT where "keep station" has the robots stay at its positions. This function, which is used at various points, is explained later in this section. Also, there is an init-delay-counter beeing increased at each time step, if the enable signal coming from the EPOS CMD interface is true (e.g. 1). This counter is used to realize a delay between completion of facility initialization (indicated by the enable signal) and transition to the next state. This delay has been introduced to ensure the values returned by the EPOS CMD interface to be valid which may not be granted instantly after facility initialization due to a certain latency in the system. Note that the substates of STATE_FACILITY_INIT are not really part of RSP server. Rather, they are part of the facility initialization process. After having started the simulation at the EPOS ACS/RT, one has to click the button "Sync" in the FMC Command Center at the FMC/MMI. EPOS moves the robots to its initial positions and attitudes ,Move to Start. Note that this has nothing to do with simulation initialization managed by RSP and has to be carried out before any commanding of the robots by a simulation on the ACS/RT is possible. As soon as this process is finished, EPOS awaits a click on "Confirm Sync" in the FMC Command Center. This click has EPOS enter synchronous commanding mode and the enable signal of the EPOS CMD interface go 1 (RSP_SVR_IN_CMD_IF_ENABLE). The init-delay-counter is increased at each time step. At some point, it exceeds a value specified by INIT_DELAY and RSP server transits to STATE_SAFE_SLOW_DOWN.

During STATE_SAFE_SLOW_DOWN the robots are decelerated to zero speed (function keep station). There are two substates. Initially, the substate ERROR_NOT_AN_ERROR is entered. As soon as the robots are standing still, the sub state machine is finished. The other substate, STATE_CMD_TOO_LATE, indicates a delayed CMD packet and is entered during simulation (see below). If the enable signal is false, which means that somethings went wrong and EPOS isn't ready for synchronous commanding any more, RSP server returns to STATE_FACILITY_INIT. Otherwise, and provided that the robots are standing still, RemoteSim-Server transits to STATE_WAIT_FOR_CONNECTION.

Keep station is executed also during STATE_WAIT_FOR_CONNECTION. Again, if RSP_SVR_IN_CMD_IF_ENABLE is false, RSP server returns to STATE_FACILITY_INIT. The next state (the large super state in Fig.2.14) is reached only if several conditions are fullfilled. As with all other states explained before, RSP_SVR_IN_CMD_IF_ENABLE must still be true. A connection must have been established with a RemoteSim-Client. The SCP layer indicates this with its output SCP_OUT_CONNECTION_OK and RSP can use this signal via its input RSP_SVR_IN_CONNECTION_OK. Moreover, valid reference time steps for client and server (inputs RSP_SVR_IN_CLIENT_REF_TIMESTEP and RSP_SVR_IN_SERVER_REF_TIMESTEP) must be provided by the SCP layer. For RSP server, SCP_OUT_HOME_REF_TIMESTEP corresponds to RSP_SVR_IN_SERVER_REF_TIMESTEP and SCP_OUT_TARGET_REF_TIMESTEP to RSP_SVR_IN_CLIENT_REF_TIMESTEP. Associated with the following transition is the activity of saving the reference time steps in internal variables for use with the subsequent simulation run.

Next, STATE_WAIT_FOR_INIT_CMD is entered. The robots are still to stay at its current positions (keep station). The server waits for a CMD_INIT packet to be received from the RSP client, which is equivalent with RSP_PACKET_CMD=CMD_INIT of RSP_SVR_IN_RECVD_RSP_PACKET. When such a packet is received, RSP server initializes the simulation and calculates an interpolation for the initial trajectory before the transition to STATE_INIT_WAIT_FOR_PUNCTUAL_CMD. Both activities are described below.

When STATE_INIT_WAIT_FOR_PUNCTUAL_CMD is active, the robots follow the initial trajectory and RSP server waits for the first CMD after the initial conditions for robot movement. In the example depicted in Fig.2.16 (e.g. Fig.2.12), RemoteSim-Server stays in this state until time step 49. RSP server will transit to STATE_INIT, if a proper CMD packet is received. Proper means that two conditions are fullfilled: RSP_PACKET_TIMESTEP has to equal next-but-one hit and RSP_PACKET_CMD

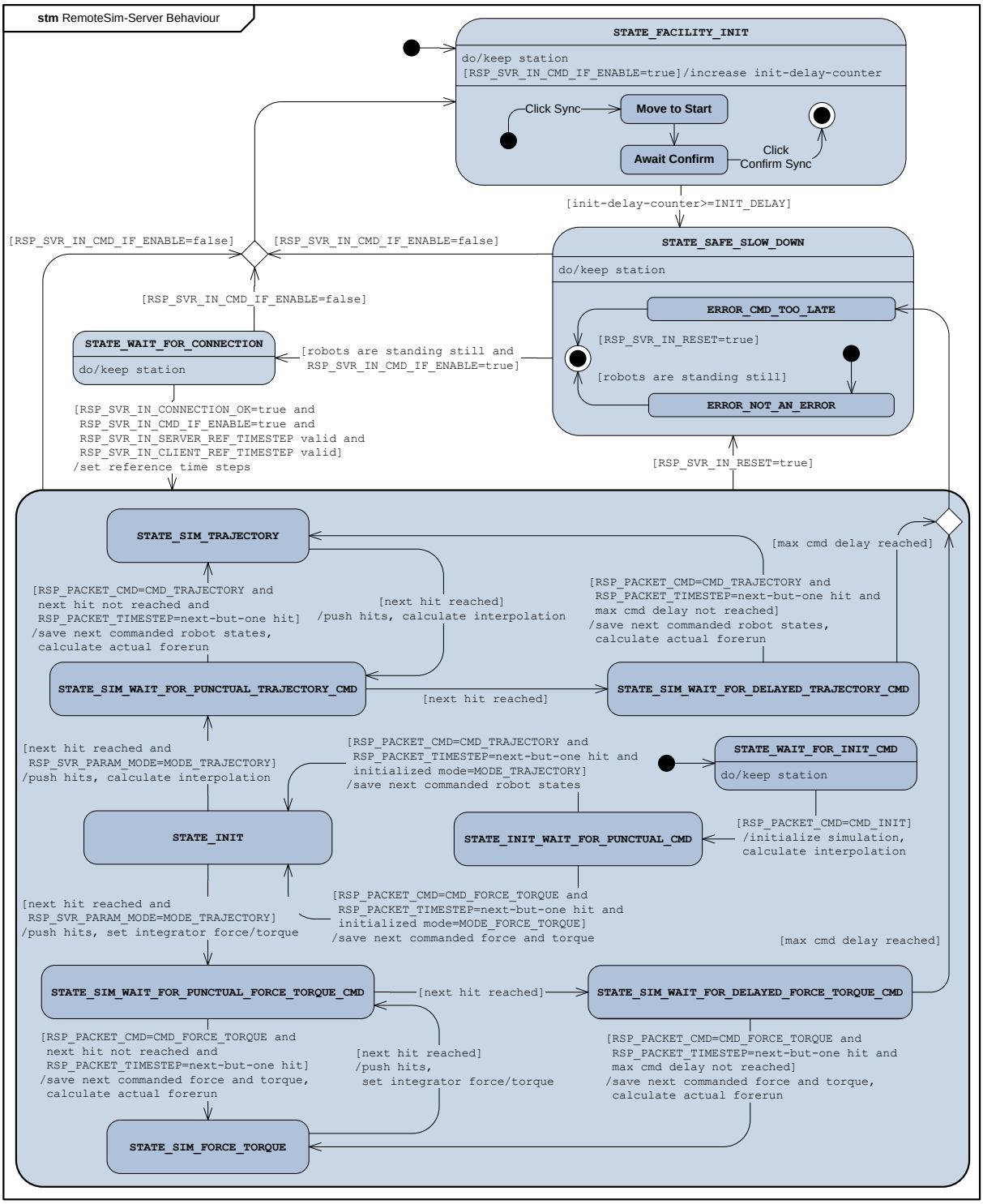

Figure 2.14.: UML state diagram of RemoteSim-Server Behaviour

must equal the type initialized with the CMD_INIT packet (CMD_TRAJECTORY or CMD_FORCE_TORQUE). All other CMD packets are ignored.

STATE_INIT is some kind of a passive state. The only action taken is to wait until next hit is reached, which represents the initial time step. At that point, there are two branches the state machine can follow, depending on whether the trajectory or force/torque is commanded. Both are widely similar, therefore they are described in parallel subsequently. If RSP server is initialized MODE_TRAJECTORY,

the next state will be STATE_SIM_WAIT_FOR_PUNCTUAL_TRAJECTORY_CMD. This is accompanied by calculation of the interpolated trajectory for the next hit interval, e.g. between next hit and next-but-one hit, and by pushing the hits (next-but-one hit becomes next hit etc., see Fig.2.12) thereafter. If RSP server is initialized MODE_FORCE_TORQUE, the next state will be STATE_SIM_WAIT_FOR_PUNCTUAL_FORCE_TORQUE_CMD. This is accompanied by pushing the hits. There is no interpolation required, since the robots' states are calculated by the integrator embedded in the ACS/RT simulation at each time step.

In principle, the states STATE_SIM_WAIT_FOR_PUNCTUAL_TRAJECTORY_CMD and STATE_SIM_WAIT_FOR_PUNCTUAL_FORCE_TORQUE_CMD are similar to the associated state of initialization STATE_INIT_WAIT_FOR_PUNCTUAL_CMD. RSP server waits for a proper CMD packet which contains the requested states to be reached at next-but-one hit. If next hit is reached and no proper packet has been received, the CMD packet is considered to be delayed and the state machine transits to STATE_SIM_WAIT_FOR_DELAYED_TRAJECTORY_CMD (or in case of force/torque commanding STATE_SIM_WAIT_FOR_DELAYED_FORCE_TORQUE_CMD). There can be no new interpolated trajectory, hence based on the old interpolation interval the trajectory is now extrapolated. However, if a proper CMD packet is received (proper meaning as in STATE_INIT_WAIT_FOR_PUNCTUAL_CMD) and next hit has not been reached yet, the requested robot states or force and torque are saved, the actual forerun ν_{actual} is calculated according to (2.18) and RSP server transits to STATE_SIM_TRAJECTORY or STATE_SIM_FORCE_TORQUE, respectively.

STATE_SIM_TRAJECTORY and STATE_SIM_FORCE_TORQUE are similar to STATE_INIT. They are essentially waiting states. As soon as next hit is reached, the trajectory for the beginning interpolation interval is calculated (trajectory commanding) or force and torque are set for the integrator (force/-torque commanding). Also, the hits are pushed. Thereupon transition back to STATE_SIM_WAIT_FOR_PUNCTUAL_TRAJECTORY_CMD (or in the case of force/torque commanding to STATE_SIM_WAIT_FOR_PUNCTUAL_FORCE_TORQUE_CMD) is conducted.

Transition from STATE_SIM_WAIT_FOR_DELAYED_TRAJECTORY_CMD to STATE_SIM_TRAJECTORY or from STATE_SIM_WAIT_FOR_PUNCTUAL_FORCE_TORQUE_CMD to STATE_SIM_FORCE_TORQUE are similar to the punctual counterparts, except one condition: Instead of next hit, the maximum command delay must not have been reached. This delay is the time passed since next hit, e.g. since the CMD packet was expected to be received at the latest. If this limit is violated, RSP server will transit to STATE_SAFE_SLOW_DOWN with substate ERROR_CMD_TOO_LATE active, which can be left only if a reset is carried out (1 at RSP_SVR_IN_RESET for at least one time step, then 0).

Thus, during simulation, if CMD packets are received in time, the state machine will oscillate between STATE_SIM_WAIT_FOR_PUNCTUAL_TRAJECTORY_CMD and STATE_SIM_TRAJECTORY (or between STATE_SIM_WAIT_FOR_PUNCTUAL_FORCE_TORQUE_CMD and STATE_SIM_FORCE_TORQUE for force/torque commanding). RSP server waits for the proper CMD packet, saves the requested values upon receipt, calculates interpolation/sets force and torque when next hit is reached, waits for the next proper CMD packet...

If, however, CMD packets are received delayed, RSP server goes from STATE_SIM_WAIT_FOR_PUNCTUAL_TRAJECTORY_CMD to its delayed pendant STATE_SIM_WAIT_FOR_DELAYED_TRAJECTORY_CMD and finally to STATE_SIM_TRAJECTORY, whereupon this series of states starts anew. The force/torque commanding equivalent is STATE_SIM_WAIT_FOR_PUNCTUAL_FORCE_TORQUE_CMD, followed by the transition to STATE_SIM_WAIT_FOR_DELAYED_FORCE_TORQUE_CMD and finally to STATE_SIM_FORCE_TORQUE. RSP server waits for the proper punctual CMD packet, then waits for a delayed CMD packet since no one has been received in time, calculates interpolation/sets force and torque instantly when the delayed packet is received, waits for the next proper CMD packet... Note that hits are pushed not before a CMD packet has been received. Therefore, during STATE_SIM_WAIT_FOR_DELAYED_TRAJECTORY_CMD or for force/torque commanding during STATE_SIM_WAIT_FOR_DELAYED_FORCE_TORQUE_CMD, next hit is smaller than the current time step.

2.6.4.3. Keep Station

In various states, the robots are to stay at its current positions. Since maximum acceleration is physically contstrained, this is equivalent with decelerating the robots to zero speed and rotation. The maximum translational and angular acceleration to apply is given by the RSP parameters $\ddot{r}_{brake}$ (RSP_SVR_PARAM_BRAKE_ACC) and $\dot{\omega}_{brake}$ (RSP_SVR_PARAM_ANGULAR_BRAKE_ACC). RSP assumes that the robots can be accelerated safely as long as these limits are not exceeded. This is a simplified

assumption of course, since joint acceleration may be considerably larger than absolute tranlational or rotational acceleration of the tool flange.

Assume that a S/C's speed for the last time step $n-1$ is given as $\left(\dot{\boldsymbol{r}}_{cmd}^{CLW}\right)_{n-1}$, then the speed at time step n yields $(i = x, y, z)$

$$\left(\dot{r}_{cmd,i}^{CLW}\right)_n = \begin{cases} \max\left[\left(\dot{r}_{cmd,i}^{CLW}\right)_{n-1} - \Delta t_{sample,s} \cdot \ddot{r}_{brake}\,,\, 0\right] & \left(\dot{r}_{cmd,i}^{CLW}\right)_{n-1} \geq 0 \\ \min\left[\left(\dot{r}_{cmd,i}^{CLW}\right)_{n-1} + \Delta t_{sample,s} \cdot \ddot{r}_{brake}\,,\, 0\right] & \left(\dot{r}_{cmd,i}^{CLW}\right)_{n-1} < 0 \end{cases} \tag{2.22}$$

and the associated position to be commanded

$$\left(\boldsymbol{r}_{cmd}^{CLW}\right)_n = \left(\boldsymbol{r}_{cmd}^{CLW}\right)_{n-1} + \frac{1}{2} \cdot \Delta t_{sample,s} \cdot \left[\left(\dot{\boldsymbol{r}}_{cmd}^{CLW}\right)_{n-1} + \left(\dot{\boldsymbol{r}}_{cmd}^{CLW}\right)_n\right] \tag{2.23}$$

Calculation of speed is trivial. To understand the latter equation, consider the general case of a moving mass point

$$r = r_0 + \int_0^1 \dot{r}(t)dt$$

The integral can be approximated (discretized) in many ways. The one chosen here (central difference scheme) is

$$r = r_o + \frac{1}{2}\Delta t\left(\dot{r}_o + \dot{r}_1\right)$$

Assume that a S/C's angular velocity for the last time step $n-1$ is given as $\left(\boldsymbol{\omega}_{cmd}^{Body}\right)_{n-1}^{CLW}$. Then the angular velocity at time step n is calculating by reducing the absolute value of each component of the angular velocity vector similar to (2.22). Thus

$$\left(\omega_{cmd,i}^{Body}\right)_n^{CLW} = \begin{cases} \max\left[\left(\omega_{cmd,i}^{Body}\right)_{n-1}^{CLW} - \Delta t_{sample,s} \cdot \dot{\omega}_{brake}\,,\, 0\right] & \left(\omega_{cmd,i}^{Body}\right)_{n-1}^{CLW} \geq 0 \\ \min\left[\left(\omega_{cmd,i}^{Body}\right)_{n-1}^{CLW} + \Delta t_{sample,s} \cdot \dot{\omega}_{brake}\,,\, 0\right] & \left(\omega_{cmd,i}^{Body}\right)_{n-1}^{CLW} < 0 \end{cases} \tag{2.24}$$

with $i = x, y, z$. Similarly to (2.23), calculate an average angular velocity

$$\left(\overline{\boldsymbol{\omega}_{cmd}^{Body}}\right)^{CLW} = \frac{1}{2} \cdot \left[\left(\boldsymbol{\omega}_{cmd}^{Body}\right)_{n-1}^{CLW} + \left(\boldsymbol{\omega}_{cmd}^{Body}\right)_n^{CLW}\right] \tag{2.25}$$

and the associated time derivative of the quaternion using the attitude at time step $n-1$ (which is again an approximation) yields according to [10, p.263]

$$\left(\overline{\dot{q}_{cmd}}\right)^{CLW} = \frac{1}{2} \cdot \left(q_{cmd}\right)_{n-1} \cdot \left(\overline{\boldsymbol{\omega}_{cmd}^{Body}}\right)^{CLW} \tag{2.26}$$

Finally, the next attitude can easily be computed

$$\left(q_{cmd}\right)_n^{CLW} = \left(q_{cmd}\right)_{n-1}^{CLW} + \Delta t_{sample,s} \cdot \left(\overline{\dot{q}_{cmd}}\right)^{CLW} \tag{2.27}$$

2.6.4.4. Initialize Simulation

The simulation run is initialized upon receipt of a `CMD_INIT` packet during `STATE_WAIT_FOR_INIT_CMD`. This packet contains information required for initialization. Using `RSP_PACKET_TIMESTEP` representing the initial time step as well as `RSP_PACKET_SAMPLETIME` representing the client sample time, next hit and next-but-one hit can be calculated. First, server client step ratio is determined according to (2.14). Then follows

$$n_{hit} = n_{init,s}^S \tag{2.28}$$

$$n_{hit+1} = n_{init,s}^S + \rho_{sample} \tag{2.29}$$

Pushing hits is achieved by increasing both next hit and next-but-one hit by ρ_{sample}.

2.6.4.5. Interpolation

RSP assumes that the sample time of RemoteSim-Client is considerably larger than that of RemoteSim-Server. The robots have to be supplied with trajectory CMDs with a frequency of $250Hz$. Reasonable frequencies for a remote simulation running satellite flight software lie in the vicinity of $10Hz$. Clearly, interpolation of the robots' trajectories in-between requested states (force/torque) is required. To avoid peaks in acceleration which the robots couldn't take and would have the facility stop the simulation, robot speed and angular velocity must be continuous. In mathematical terms the interpolation must be C^1 continuous for both position and attitude. In general, interpolation starts at time t^S_{start} and ends at time t^S_{end}. During regular simulation (CMD packets coming in time) interpolation starts at t^S_{hit} and ends at t^S_{hit+1}. Only if CMD packets are received delayed does t^S_{start} exceed next hit. Translational and rotational interpolation are treated separately in this section, beginning with the former.

Interpolation of position. Position is a vector of three components. Interpolation can be carried out for each component independently. That's why interpolation is given here for a scalar curve which must then be applied to each component. To comply with C^1 continuity, the interpolated curve must be consistent with the following constraints. The curve must start at position r^{CLW}_{start} with speed $\dot{r}^{CLW}_{start}$ at time t^S_{start} and end at position r^{CLW}_{end} with speed $\dot{r}^{CLW}_{end}$ at time t^S_{end}. For simplicity, the author chooses a polynomial as ansatz. The most simple polynomial which allows all boundary conditions to be included is of third order.

$$r(t) = at^3 + bt^2 + ct + d \tag{2.30}$$

The associated ansatz for robot speed is

$$\dot{r}(t) = 3at^2 + 2bt + c \tag{2.31}$$

Inserting the boundary conditions yields

$$r^{CLW}_{start} = a\left(t^S_{start}\right)^3 + b\left(t^S_{start}\right)^2 + c\left(t^S_{start}\right) + d \tag{2.32}$$

$$r^{CLW}_{end} = a\left(t^S_{end}\right)^3 + b\left(t^S_{end}\right)^2 + c\left(t^S_{end}\right) + d \tag{2.33}$$

$$\dot{r}^{CLW}_{start} = 3a\left(t^S_{start}\right)^2 + 2b\left(t^S_{start}\right) + c \tag{2.34}$$

$$\dot{r}^{CLW}_{end} = 3a\left(t^S_{end}\right)^2 + 2b\left(t^S_{end}\right) + c \tag{2.35}$$

or in matrix form

$$\begin{pmatrix} r^{CLW}_{start} \\ r^{CLW}_{end} \\ \dot{r}^{CLW}_{start} \\ \dot{r}^{CLW}_{end} \end{pmatrix} = \begin{pmatrix} \left(t^S_{start}\right)^3 & \left(t^S_{start}\right)^2 & \left(t^S_{start}\right) & 1 \\ \left(t^S_{end}\right)^3 & \left(t^S_{end}\right)^2 & \left(t^S_{end}\right) & 1 \\ 3\left(t^S_{start}\right)^2 & 2\left(t^S_{start}\right) & 1 & 0 \\ 3\left(t^S_{end}\right)^2 & 2\left(t^S_{end}\right) & 1 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} \tag{2.36}$$

The linear equation system (2.36) is solved by Gaussian elimination (see [15, p.93]). The solution is the parameters a, b, c, d which realize $r^S_{cmd}(t)$ with $t \in \left[t^S_{start}, t^S_{end}\right]$ for interpolation and $t \in \left]t^S_{end}, \infty\right[$ for extrapolation (in the case of a delayed CMD packet).

Interpolation of attitude. Interpolation of attitude as a quaternion is more complicated. Interpolation must not take place in Euclidean space R^3. Example: Simply interpolating the single components of a quaternion will most likely lead to non-unit quaternions which don't represent a valid rotation at all. Rather interpolation of attitude must take place in the rotation group $SO(3)$ (see [15, p.107]). The method RSP uses is adopted from Kim, Kim, Shin (see [16]). It is based on applying principles of spline curves, like hermite, to construct unit quaternion curves with similar properties. The reader is encouraged to study [15] for further details. Here, the essential equations for realizing a hermite quaternion interpolation are given, consistent with the notation in this thesis. Note that in [15] the length of an interpolated curve referes to a timespan of 1 (whatever the

unit). Therefore, the following equations contain modifications for having an interpolation of duration $\Delta t_{interp} = t^S_{end} - t^S_{start}$. Similar to interpolation of position, C^1 continuity is required. The quaternion curve must start at attitude $(q_{start})^{CLW}$ with angular velocity $\left(\boldsymbol{\omega}^{Body}_{start}\right)^{CLW}$ at time t^S_{start} and end at attitude $(q_{end})^{CLW}$ with angular velocity $\left(\boldsymbol{\omega}^{Body}_{end}\right)^{CLW}$ at time t^S_{end}.

First, some definitions are required.

Given a vector $\boldsymbol{v} = \theta\hat{\boldsymbol{v}}$, the exponential for an unit quaternion is defined

$$\exp(\boldsymbol{v}) = \cos(\theta) + \hat{\boldsymbol{v}}\sin(\theta) \tag{2.37}$$

So the exponential takes a vector as an argument and returns an unit quaternion.

Given a quaternion $q = \cos(\theta) + \hat{\boldsymbol{v}}\sin(\theta)$, the logarithm for an unit quaternion is defined

$$\log(q) = \theta\hat{\boldsymbol{v}} \tag{2.38}$$

So the logarithm takes an unit quaternion as an argument and returns a vector.

The Bernstein basis yields

$$\beta_{i,k}(s) = \binom{k}{i}(1-s)^{k-i}t^i \tag{2.39}$$

The cumulative Bernstein basis can be obtained

$$\tilde{\beta}_{j,k}(s) = \sum_{i=j}^{k}\beta_{i,k}(s) \tag{2.40}$$

Here, the following cumulative bases are important:

$$\tilde{\beta}_{1,3}(s) = 1 - (1-s)^3 \tag{2.41}$$
$$\tilde{\beta}_{2,3}(s) = 3s^2 - 2s^3 \tag{2.42}$$
$$\tilde{\beta}_{3,3}(s) = s^3 \tag{2.43}$$
$$\tag{2.44}$$

Then the hermite quaternion curve is given by

$$(q_{cmd})^{CLW}(s) = (q_{start})^{CLW}\prod_{j=1}^{3}\exp\left(\omega_j\tilde{\beta}_{j,3}(s)\right) \tag{2.45}$$

where

$$\omega_1 = \exp\left(\frac{\Delta t_{interp}\cdot\left(\boldsymbol{\omega}^{Body}_{start}\right)^{CLW}}{6}\right) \tag{2.46}$$

$$\omega_2 = \log\left[\exp\left(\frac{\Delta t_{interp}\cdot\left(\boldsymbol{\omega}^{Body}_{start}\right)^{CLW}}{6}\right)^{-1}\cdot(q_{start})^{CLW}\cdot(q_{end})^{CLW}\cdot \cdot\exp\left(\frac{\Delta t_{interp}\cdot\left(\boldsymbol{\omega}^{Body}_{end}\right)^{CLW}}{6}\right)^{-1}\right] \tag{2.47}$$

$$\omega_3 = \exp\left(\frac{\Delta t_{interp}\cdot\left(\boldsymbol{\omega}^{Body}_{end}\right)^{CLW}}{6}\right) \tag{2.48}$$

The scale factor s allows to address a specific point of the interpolated curve, $s = 0$ corresponding to $(q_{start})^{CLW}$ and $s = 1$ to $(q_{end})^{CLW}$. Hence

$$s = \frac{t^S - t^S_{start}}{\Delta t_{interp}} \tag{2.49}$$

is used to return the interpolated quaternion at t^S. Note that extrapolation is also possible with $s > 1$.

2.6.4.6. Coordinate Transformation

As outlined in Sec.2.1, the RSP concept provides for the user two possibilities of supplying the robots with trajectory CMDs.

- Feedthrough
- Earth-Center-Inertial (ECI) to Clohessy-Wiltshire (CLW) Conversion

The former is trivial. All move commands are fed through directly to the RemoteSim-Server and appear (interpolated) at its outputs. The latter demands the user to provide RSP server with move commands expressed in the ECI frame. Also, RSP client returns the actual robot states expressed in these coordinates. Thus, the user has only his own view of the whole simulation chain - as if the robots would be moving in space - and he doesn't have to deal with any further processing. The calculations are then carried out by RSP, invisible to the user.

For the subsequent derivation, an additonal coordinate system is used: The Shifted-Earth-Center-Inertial (SECI) frame. As the name suggests, this is simply a translationally shifted ECI system. It seems to be uncommon to use such a system for clarity in illustrating coordinate transformations. But just for this reason the author chooses to use it at this point.

The transformation is carried out in two steps. First, ECI is shifted translationally into SECI. That's why SECI has been introduced as an auxillary coordinate system. Second, SECI is turned into CLW.

Given some position vector $\mathbf{r}^{ECI}$ in ECI-frame, its representation in SECI coordinates can be obtained by adding some vector representing the shift from ECI to SECI

$$\mathbf{r}^{SECI} = \mathbf{r}^{ECI} + \mathbf{r}^{ECI \to SECI} \tag{2.50}$$

For now, this translational vector is unknown and will be determined later. At any rate, it is clear that such a vector exists and we can work with it at this point.

As the second step, assume that SECI is turned into CLW through angle $\varphi^{SECI \to CLW}$ about axis $\mathbf{a}^{SECI \to CLW}$. It's subtle but important to realize that $\mathbf{a}^{SECI \to CLW}$ is given in SECI coordinates. Angle and axis define a unit quaternion

$$q^{SECI \to CLW} = \cos\left(\frac{\varphi^{SECI \to CLW}}{2}\right) + \mathbf{a}^{SECI \to CLW} \sin\left(\frac{\varphi^{SECI \to CLW}}{2}\right) \tag{2.51}$$

It can now be used to transform any vector from SECI to CLW frame by an according rotation.

$$r^{CLW} = \left(q^{SECI \to CLW}\right)^{\star} r^{SECI} q^{SECI \to CLW} \tag{2.52}$$

Combining 2.50 and 2.52 yields

$$r^{CLW} = \left(q^{SECI \to CLW}\right)^{\star} \left(r^{ECI} + r^{ECI \to SECI}\right) q^{SECI \to CLW} \tag{2.53}$$

To get the inverse of the transformation, (2.53) is rearranged as follows

$$q^{SECI \to CLW} r^{CLW} = \left(r^{ECI} + r^{ECI \to SECI}\right) q^{SECI \to CLW}$$

$$q^{SECI \to CLW} r^{CLW} \left(q^{SECI \to CLW}\right)^{\star} = r^{ECI} + r^{ECI \to SU}$$

and finally

$$r^{ECI} = q^{SECI \to CLW} r^{CLW} \left(q^{SECI \to CLW}\right)^{\star} - r^{ECI \to SECI} \tag{2.54}$$

Calculating the transformed attitude is even simpler. Assume $q^{SECI \to CLW}$ is given, as well as the spacecraft's attitude in ECI-frame q^{ECI}. Since attitude is invariant to a translational shift of the coordinate system, the expression

$$q^{SECI} = q^{ECI} \tag{2.55}$$

holds. This means the spacecraft's attitude in SECI-frame is known. Now, SECI-frame is turned into CLW-frame. This means, that also q^{SECI} is turned through the same angle about the same axis. This can be achieved by simply multiplying q^{SECI} by $q^{SECI \to CLW}$ "from the left" (!), see [10, p.134].

$$q^{CLW} = q^{SECI \to CLW} q^{SECI} \tag{2.56}$$

Thus, the simple attitude transformation yields

$$q^{CLW} = q^{SECI \to CLW} q^{ECI} \tag{2.57}$$

and its reverse respectively

$$q^{ECI} = \left(q^{SECI \to CLW}\right)^{\star} q^{CLW} \tag{2.58}$$

In conclusion, the transformation $CLW \leftrightarrow ECI$ of position and attitude is unambiguously defined by the set $(\mathbf{r}^{ECI \to SECI}, q^{SECI \to CLW})$.

The next step is to compute this set.

Consider the relation between ECI and CLW frame. The CLW frame's origin constitutes some in-orbit reference point. Here, this point equals the target spacecraft's position $\boldsymbol{r}_t^{ECI}$. The CLW z-axis $\hat{\boldsymbol{Z}}_{CLW}$ points towards the ECI origin, that is the earth's center. The CLW frame's x-axis $\hat{\boldsymbol{X}}_{CLW}$ lies in the plane the CLW frame's z-axis and target's velocity vector $\dot{\boldsymbol{r}}_t^{ECI}$ span. Finally, the CLW y-axis $\hat{\boldsymbol{Y}}_{CLW}$ completes a right-handed-side coordinate system.

By definition, the CLW frame's origin coincides with the target spacecraft. This means that

$$\mathbf{r}_t^{CLW} = \mathbf{0} \tag{2.59}$$

and with (2.52)

$$r_t^{SECI} = q^{SECI \to CLW} r_t^{CLW} \left(q^{SECI \to CLW}\right)^{\star} = q^{SECI \to CLW} 0 \left(q^{SECI \to CLW}\right)^{\star} = 0 \tag{2.60}$$

holds. Finally, applying (2.50) yields

$$\mathbf{r}^{ECI \to SECI} = \mathbf{r}_t^{SECI} - \mathbf{r}_t^{ECI} = -\mathbf{r}_t^{ECI} \tag{2.61}$$

The transformation quaternion is determined by making use of the associated rotation matrix. First, consider the CLW coordinate axis expressed as unit vectors in ECI coordinates. The z-axis is simply the negative of the target spacecraft's position vector (normalized).

$$\hat{\boldsymbol{Z}}_{CLW}^{ECI} = -\frac{\boldsymbol{r}_t^{ECI}}{\left|\boldsymbol{r}_t^{ECI}\right|} = -\hat{\boldsymbol{r}}_t^{ECI} \tag{2.62}$$

The y-axis is determined by computing the vector product of the CLW z-axis and the target spacecraft's normalized velocity vector.

$$\hat{\boldsymbol{Y}}_{CLW}^{ECI} = \hat{\boldsymbol{Z}}_{CLW}^{ECI} \times \frac{\dot{\boldsymbol{r}}_t^{ECI}}{\left|\dot{\boldsymbol{r}}_t^{ECI}\right|} = \hat{\boldsymbol{Z}}_{CLW}^{ECI} \times \hat{\dot{\boldsymbol{r}}}_t^{ECI} \tag{2.63}$$

And finally to complete the right-handed-side coordinate system

$$\hat{\boldsymbol{X}}_{CLW}^{ECI} = \hat{\boldsymbol{Y}}_{CLW}^{ECI} \times \hat{\boldsymbol{Z}}_{CLW}^{ECI} \tag{2.64}$$

Similar to the transformation quaternion, denote the transformation matrix $M^{CLW \to SECI}$. Note that the direction of transformation is opposite to that of the quaternion introduced earlier. The above calculated unit vectors constitute the matrix' columns.

$$M^{CLW \to SECI} = \begin{pmatrix} \hat{\boldsymbol{X}}_{CLW}^{ECI} & \hat{\boldsymbol{Y}}_{CLW}^{ECI} & \hat{\boldsymbol{Z}}_{CLW}^{ECI} \end{pmatrix} = \begin{pmatrix} X_{x,CLW}^{ECI} & Y_{x,CLW}^{ECI} & Z_{x,CLW}^{ECI} \\ X_{y,CLW}^{ECI} & Y_{y,CLW}^{ECI} & Z_{y,CLW}^{ECI} \\ X_{z,CLW}^{ECI} & Y_{z,CLW}^{ECI} & Z_{z,CLW}^{ECI} \end{pmatrix} \tag{2.65}$$

To clarify the notation, here an example: $Y^{ECI}_{z,CLW}$ is the z-component of the CLW frame's y-axis unit vector expressed in ECI coordinates. This matrix realizes the transformation

$$\boldsymbol{r}^{SECI} = M^{CLW \to SECI} \boldsymbol{r}^{CLW} \tag{2.66}$$

which corresponds to

$$r^{SECI} = \left(q^{CLW \to SECI}\right)^{\star} r^{CLW} q^{CLW \to SECI} \tag{2.67}$$

Accordingly the transformation

$$\boldsymbol{r}^{CLW} = \left(M^{CLW \to SECI}\right)^{-1} \boldsymbol{r}^{SECI} = \left(M^{CLW \to SECI}\right)^{T} \boldsymbol{r}^{SECI} \tag{2.68}$$

corresponds to

$$r^{CLW} = \left(q^{SECI \to CLW}\right)^{\star} r^{SECI} q^{SECI \to CLW} \tag{2.69}$$

From that follows the rotation matrix which corresponds to the unit quaternion $q^{SECI \to CLW}$.

$$M^{SECI \to CLW} = \left(M^{CLW \to SECI}\right)^{T} = \begin{pmatrix} X^{ECI}_{x,CLW} & X^{ECI}_{y,CLW} & X^{ECI}_{z,CLW} \\ Y^{ECI}_{x,CLW} & Y^{ECI}_{y,CLW} & Y^{ECI}_{z,CLW} \\ Z^{ECI}_{x,CLW} & Z^{ECI}_{y,CLW} & Z^{ECI}_{z,CLW} \end{pmatrix} \tag{2.70}$$

In general, given some rotation matrix

$$M = \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix}$$

the unit quaternion with scalar part q_0 and vector part $\boldsymbol{q} = (q_1 \quad q_2 \quad q_3)^T$ representing the same rotation can be calculated as follows (see [10, p.159]):

$$\begin{aligned} q_0 &= \frac{1}{2}\sqrt{m_{00} + m_{11} + m_{22} + 1} \\ q_1 &= \frac{(m_{12} - m_{21})}{4q_0} \\ q_2 &= \frac{(m_{20} - m_{02})}{4q_0} \\ q_3 &= \frac{(m_{01} - m_{10})}{4q_0} \end{aligned}$$

In this way, the unit quaternion $q^{SECI \to CLW}$ can be computed from the rotation matrix $M^{SECI \to CLW}$.

$$q_0^{SECI \to CLW} = \frac{1}{2}\sqrt{X^{ECI}_{x,CLW} + Y^{ECI}_{y,CLW} + Z^{ECI}_{z,CLW} + 1} \tag{2.71}$$

$$q_1^{SECI \to CLW} = \frac{(Y^{ECI}_{z,CLW} - Z^{ECI}_{y,CLW})}{4q_0} \tag{2.72}$$

$$q_2^{SECI \to CLW} = \frac{(Z^{ECI}_{x,CLW} - X^{ECI}_{z,CLW})}{4q_0} \tag{2.73}$$

$$q_3^{SECI \to CLW} = \frac{(X^{ECI}_{y,CLW} - Y^{ECI}_{x,CLW})}{4q_0} \tag{2.74}$$

Interpolation of robot positions, as realized with RemoteSim, also requires speed and angular velocity. Therefore, the coordinate transformation has to include these values too. The approach is somewhat different.

The speed of some spacecraft can be written as

$$\left(\dot{\boldsymbol{r}}^{ECI}\right)^{ECI} = \left(\dot{\boldsymbol{r}}^{ECI}_{SECI}\right)^{ECI} + \left(\dot{\boldsymbol{r}}^{ECI}\right)^{SECI} \tag{2.75}$$

The speed is the sum of the moving SECI frame's speed and the spacecraft's speed with respect to this moving system. Speed with respect to SECI can be obtained by considering the first derivative in a rotating coordinate system (see [11, p.274]).

Remembering that the origins of CLW and SECI system coincide and that ECI is an inertial frame, the expression

$$\left(\dot{\boldsymbol{r}}^{CLW}\right)^{SECI} = \left(\dot{\boldsymbol{r}}^{CLW}\right)^{CLW} + \left(\boldsymbol{\omega}^{CLW}_{CLW}\right)^{SECI} \times \boldsymbol{r}^{CLW} \tag{2.76}$$

holds. This speed vector can be converted into SECI coordinates using the transformation quaternion computed above.

$$\left(\dot{\boldsymbol{r}}^{SECI}\right)^{SECI} = q^{SECI\to CLW}\left(\dot{\boldsymbol{r}}^{CLW}\right)^{SECI}\left(q^{SECI\to CLW}\right)^{\star} \tag{2.77}$$

And finally, considering that

$$\left(\dot{\boldsymbol{r}}^{ECI}\right)^{SECI} = \left(\dot{\boldsymbol{r}}^{SECI}\right)^{SECI} \tag{2.78}$$

it follows

$$\begin{aligned}\left(\dot{\boldsymbol{r}}^{ECI}\right)^{ECI} &= \left(\dot{\boldsymbol{r}}^{ECI}_{SECI}\right)^{ECI} + \left(\dot{\boldsymbol{r}}^{SECI}\right)^{SECI} \\ &= \left(\dot{\boldsymbol{r}}^{ECI}_{SECI}\right)^{ECI} + q^{SECI\to CLW}\left(\dot{\boldsymbol{r}}^{CLW}\right)^{SECI}\left(q^{SECI\to CLW}\right)^{\star} \\ &= \left(\dot{\boldsymbol{r}}^{ECI}_{SECI}\right)^{ECI} + q^{SECI\to CLW}\left[\left(\dot{\boldsymbol{r}}^{CLW}\right)^{CLW} + \left(\boldsymbol{\omega}^{CLW}_{CLW}\right)^{SECI} \times \boldsymbol{r}^{CLW}\right]\left(q^{SECI\to CLW}\right)^{\star}\end{aligned} \tag{2.79}$$

The reverse transformation is of course

$$\left(\dot{\boldsymbol{r}}^{CLW}\right)^{CLW} = \left(q^{SECI\to CLW}\right)^{\star}\left[\left(\dot{\boldsymbol{r}}^{ECI}\right)^{ECI} - \left(\dot{\boldsymbol{r}}^{ECI}_{SECI}\right)^{ECI}\right]q^{SECI\to CLW} - \left(\boldsymbol{\omega}^{CLW}_{CLW}\right)^{SECI} \times \boldsymbol{r}^{CLW} \tag{2.80}$$

$\left(\dot{\boldsymbol{r}}^{CLW}\right)^{CLW}$ is the spacecraft's speed with respect to CLW, expressed in CLW coordinates. $\left(\dot{\boldsymbol{r}}^{ECI}\right)^{ECI}$ is the spacecraft's speed with respect to ECI, expressed in ECI coordinates. $\left(\dot{\boldsymbol{r}}^{ECI}_{SECI}\right)^{ECI}$ is the SECI frame's speed with respect to ECI, expressed in ECI coordinates. $\left(\boldsymbol{\omega}^{SECI}_{CLW}\right)^{SECI}$ is the CLW frame's angular speed with respect to SECI, expressed in SECI coordinates. $\boldsymbol{r}^{CLW}$ is the spacecraft's position expressed in CLW coordinates. Speed of SECI and angular speed of CLW are yet to be determined.

The SECI frame's origin coincides with the target spacecraft's position. Hence

$$\left(\dot{\boldsymbol{r}}^{ECI}_{SECI}\right)^{ECI} = \left(\dot{\boldsymbol{r}}^{ECI}_{t}\right)^{ECI} \tag{2.81}$$

Since, by definition, the CLW z-axis always points to the ECI origin and the rotation vector is invariant to translational shifts of the coordinate system, $\left(\boldsymbol{\omega}^{SECI}_{CLW}\right)^{SECI} = \left(\boldsymbol{\omega}^{ECI}_{CLW}\right)^{ECI}$ must hold. Moreover, since the CLW origin cooincides with the target S/C, the angular velocity of CLW frame can easily be determined.

$$\left(\boldsymbol{\omega}^{SECI}_{CLW}\right)^{SECI} = \left(\boldsymbol{\omega}^{ECI}_{CLW}\right)^{ECI} = \frac{\boldsymbol{r}^{ECI}_{t} \times \left(\dot{\boldsymbol{r}}^{ECI}_{t}\right)^{ECI}}{\left|\boldsymbol{r}^{ECI}_{t}\right|^{2}} \tag{2.82}$$

Angular velocity of some S/C must adhere to the trivial relation

$$\left(\boldsymbol{\omega}^{SECI}\right)^{SECI} = \left(\boldsymbol{\omega}^{SECI}_{CLW}\right)^{SECI} + \left(\boldsymbol{\omega}^{SECI}\right)^{CLW} \tag{2.83}$$

A spacecraft's angular velocity with respect to the inertial frame (SECI) is the sum of the rotating frame's (CLW) angular velocity and the spacecraft's angular velocity with respect to the rotating frame. From that follows

$$\begin{aligned}\left(\omega^{CLW}\right)^{CLW} &= \left(q^{SECI\to CLW}\right)^{\star}\left(\omega^{SECI}\right)^{CLW} q^{SECI\to CLW} \\ &= \left(q^{SECI\to CLW}\right)^{\star}\left(\left(\omega^{SECI}\right)^{SECI} - \left(\omega_{CLW}^{SECI}\right)^{SECI}\right) q^{SECI\to CLW} \\ &= \left(q^{SECI\to CLW}\right)^{\star}\left(\left(\omega^{ECI}\right)^{ECI} - \left(\omega_{CLW}^{SECI}\right)^{SECI}\right) q^{SECI\to CLW}\end{aligned} \tag{2.84}$$

and

$$\left(\omega^{ECI}\right)^{ECI} = q^{SECI\to CLW}\left(\omega^{CLW}\right)^{CLW}\left(q^{SECI\to CLW}\right)^{\star} + \left(\omega_{CLW}^{SECI}\right)^{SECI} \tag{2.85}$$

In **summary**, the subsequent steps are necessary for the transformation.

1) Calculate the translational transformation vector:

$$\mathbf{r}^{ECI\to SECI} = -\mathbf{r}_t^{ECI} \tag{2.61}$$

2) Calculate the CLW axes unit vectors:

$$\hat{\boldsymbol{Z}}_{CLW}^{ECI} = -\hat{\boldsymbol{r}}_t^{ECI} \tag{2.62}$$

$$\hat{\boldsymbol{Y}}_{CLW}^{ECI} = \hat{\boldsymbol{Z}}_{CLW}^{ECI} \times \hat{\boldsymbol{r}}_t^{ECI} \tag{2.63}$$

$$\hat{\boldsymbol{X}}_{CLW}^{ECI} = \hat{\boldsymbol{Y}}_{CLW}^{ECI} \times \hat{\boldsymbol{Z}}_{CLW}^{ECI} \tag{2.64}$$

3) Calculate the rotation quaternion:

$$q_0^{SECI\to CLW} = \frac{1}{2}\sqrt{X_{x,CLW}^{ECI} + Y_{y,CLW}^{ECI} + Z_{z,CLW}^{ECI} + 1} \tag{2.71}$$

$$q_1^{SECI\to CLW} = \frac{(Y_{z,CLW}^{ECI} - Z_{y,CLW}^{ECI})}{4q_0} \tag{2.72}$$

$$q_2^{SECI\to CLW} = \frac{(Z_{x,CLW}^{ECI} - X_{z,CLW}^{ECI})}{4q_0} \tag{2.73}$$

$$q_3^{SECI\to CLW} = \frac{(X_{y,CLW}^{ECI} - Y_{x,CLW}^{ECI})}{4q_0} \tag{2.74}$$

4) Calculate the CLW frame's angular velocity:

$$\left(\boldsymbol{\omega}_{CLW}^{SECI}\right)^{SECI} = \frac{\boldsymbol{r}_t^{ECI} \times \left(\dot{\boldsymbol{r}}_t^{ECI}\right)^{ECI}}{\left|\boldsymbol{r}_t^{ECI}\right|^2} \tag{2.82}$$

5) Calculate the SECI frame's speed:

$$\left(\dot{\boldsymbol{r}}_{SECI}^{ECI}\right)^{ECI} = \left(\dot{\boldsymbol{r}}_t^{ECI}\right)^{ECI} \tag{2.81}$$

6) The transformation $ECI \to CLW$ then yields:

$$r^{CLW} = \left(q^{SECI\to CLW}\right)^{\star}\left(r^{ECI} + r^{ECI\to SECI}\right) q^{SECI\to CLW} \tag{2.53}$$

$$q^{CLW} = q^{SECI\to CLW} q^{ECI} \tag{2.57}$$

$$\left(\dot{\boldsymbol{r}}^{CLW}\right)^{CLW} = \left(q^{SECI\to CLW}\right)^{\star}\left[\left(\dot{\boldsymbol{r}}^{ECI}\right)^{ECI} - \left(\dot{\boldsymbol{r}}_{SECI}^{ECI}\right)^{ECI}\right] q^{SECI\to CLW} - \left(\boldsymbol{\omega}_{CLW}^{CLW}\right)^{SECI} \times \boldsymbol{r}^{CLW} \tag{2.86}$$

$$\left(\omega^{CLW}\right)^{CLW} = \left(q^{SECI\to CLW}\right)^{\star}\left(\left(\omega^{ECI}\right)^{ECI} - \left(\omega_{CLW}^{SECI}\right)^{SECI}\right) q^{SECI\to CLW} \tag{2.84}$$

7) The transformation $CLW \rightarrow ECI$ yields respectively:

$$r^{ECI} = q^{SECI \rightarrow CLW} r^{CLW} \left(q^{SECI \rightarrow CLW}\right)^{\star} - r^{ECI \rightarrow SECI} \tag{2.54}$$

$$q^{ECI} = \left(q^{SECI \rightarrow CLW}\right)^{\star} q^{CLW} \tag{2.58}$$

$$\left(\dot{r}^{ECI}\right)^{ECI} = \left(\dot{r}^{ECI}_{SECI}\right)^{ECI} + q^{SECI \rightarrow CLW} \left[\left(\dot{r}^{CLW}\right)^{CLW} + \left(\omega^{CLW}_{CLW}\right)^{SECI} \times r^{CLW}\right] \left(q^{SECI \rightarrow CLW}\right)^{\star} \tag{2.87}$$

$$\left(\omega^{ECI}\right)^{ECI} = q^{SECI \rightarrow CLW} \left(\omega^{CLW}\right)^{CLW} \left(q^{SECI \rightarrow CLW}\right)^{\star} + \left(\omega^{SECI}_{CLW}\right)^{SECI} \tag{2.85}$$

2.6.4.7. RSP Server Output Behaviour

While the fields of the RSP client's output packet `RSP_CLI_OUT_SEND_RSP_PACKET` have different meanings depending on current client state, this is not the case for `RSP_SVR_OUT_SEND_RSP_PACKET`.

Table 2.12.: Fields of `RSP_SVR_OUT_SEND_RSP_PACKET` (RSP Server)

`RSP_PACKET_TIMESTEP`	N/M
`RSP_PACKET_CHASER_POS`	$r^{U}_{curr,c}$
`RSP_PACKET_CHASER_SPEED`	$\dot{r}^{U}_{curr,c}$
`RSP_PACKET_CHASER_ATT`	$(q_{curr,c})^{U}$
`RSP_PACKET_CHASER_RATE`	$\left(\omega^{U}_{curr,c}\right)^{U}$
`RSP_PACKET_TARGET_POS`	$r^{U}_{curr,t}$
`RSP_PACKET_TARGET_SPEED`	$\dot{r}^{U}_{curr,t}$
`RSP_PACKET_TARGET_ATT`	$(q_{curr,t})^{U}$
`RSP_PACKET_TARGET_RATE`	$\left(\omega^{U}_{curr,t}\right)^{U}$
`RSP_PACKET_CMD`	N/M
`RSP_PACKET_COSY`	as initialized by `CMD_INIT` packet, `COSY_NOT_A_COSY` before initialization
`RSP_PACKET_MODE`	as initialized by `CMD_INIT` packet, `MODE_NOT_A_MODE` before initialization
`RSP_PACKET_ERROR`	curent server error
`RSP_PACKET_SAMPLETIME`	$\Delta t_{sample,s}$

The time step transmitted has no meaning at all. Chaser and target states always refer to the current robot positions and attitudes as returned by the EPOS CMD interface with calculated speeds and angular velocities. These values are always expressed in the coordinate system the robots are commanded. If `RSP_CLI_PARAM_COSY` is `COSY_ECI_2_CLW`, then RSP carries out the reverse coordinate transformation, such that the returned current spacecraft states are expressed in the right coordinate system and the user "sees" only the spacecraft in the User frame. The field `RSP_PACKET_CMD` has no meaning in `RSP_SVR_OUT_SEND_RSP_PACKET`. Both, `RSP_PACKET_COSY` and `RSP_PACKET_MODE` contain the values as they have been intialized by the `CMD_INT` packet from RSP client. Before initialization, the fields equal `COSY_NOT_A_COSY` and `MODE_NOT_A_MODE`. `RSP_PACKET_ERROR` always contains the

current server error and the field RSP_PACKET_SAMPLETIME equals the sample time specified as RSP parameter (RSP_SVR_PARAM_SAMPLETIME).

RSP server includes a large number of outputs. Tabs.2.13, 2.15 and 2.14 summarize the values of these outputs, in some cases depending on server state and/or other inputs.

Tab.2.13 shows outputs which indicate current and requested states of chaser and target, both in CLW and U frame. Moreover, commanded positions and attitudes are included.

Table 2.13.: Values of RSP server outputs (current chaser/target states, requested chaser/target states, commanded chaser/target positions and attitudes) depending on current server state. N/M - no meaning.

	STATE_FACILITY_INIT	STATE_SAFE_SLOW_DOWN STATE_WAIT_FOR_CONNECTION STATE_WAIT_FOR_INIT_CMD	STATE_INIT_WAIT_FOR_PUNCTUAL_CMD STATE_INIT	STATE_SIM_WAIT_FOR_PUNCTUAL_TRAJECTORY_CMD STATE_SIM_WAIT_FOR_DELAYED_TRAJECTORY_CMD STATE_SIM_TRAJECTORY	STATE_SIM_WAIT_FOR_PUNCTUAL_FORCE_TORQUE_CMD STATE_SIM_WAIT_FOR_DELAYED_FORCE_TORQUE_CMD STATE_SIM_FORCE_TORQUE
RSP_SVR_OUT_CHASER_CURR_POS_CLW	N/M	$\boldsymbol{r}_{curr,c}^{CLW}$			
RSP_SVR_OUT_CHASER_CURR_SPEED_CLW	N/M	$\dot{\boldsymbol{r}}_{curr,c}^{CLW}$			
RSP_SVR_OUT_CHASER_CURR_ATT_CLW	N/M	$(q_{curr,c})^{CLW}$			
RSP_SVR_OUT_CHASER_CURR_RATE_CLW	N/M	$\left(\boldsymbol{\omega}_{curr,c}^{Body}\right)^{CLW}$			
RSP_SVR_OUT_TARGET_CURR_POS_CLW	N/M	$\boldsymbol{r}_{curr,t}^{CLW}$			
RSP_SVR_OUT_TARGET_CURR_SPEED_CLW	N/M	$\dot{\boldsymbol{r}}_{curr,t}^{CLW}$			
RSP_SVR_OUT_TARGET_CURR_ATT_CLW	N/M	$(q_{curr,t})^{CLW}$			
RSP_SVR_OUT_TARGET_CURR_RATE_CLW	N/M	$\left(\boldsymbol{\omega}_{curr,t}^{Body}\right)^{CLW}$			
RSP_SVR_OUT_CHASER_CURR_POS_U	N/M	$\boldsymbol{r}_{curr,c}^{U}$			
RSP_SVR_OUT_CHASER_CURR_SPEED_U	N/M	$\dot{\boldsymbol{r}}_{curr,c}^{U}$			
RSP_SVR_OUT_CHASER_CURR_ATT_U	N/M	$(q_{curr,c})^{U}$			
RSP_SVR_OUT_CHASER_CURR_RATE_U	N/M	$\left(\boldsymbol{\omega}_{curr,c}^{Body}\right)^{U}$			
RSP_SVR_OUT_TARGET_CURR_POS_U	N/M	$\boldsymbol{r}_{curr,t}^{U}$			
RSP_SVR_OUT_TARGET_CURR_SPEED_U	N/M	$\dot{\boldsymbol{r}}_{curr,t}^{U}$			

Continued on next page

Table 2.13 – *Continued from previous page*

	STATE_FACILITY_INIT	STATE_SAFE_SLOW_DOWN STATE_WAIT_FOR_CONNECTION STATE_WAIT_FOR_INIT_CMD	STATE_INIT_WAIT_FOR_PUNCTUAL_CMD STATE_INIT	STATE_SIM_WAIT_FOR_PUNCTUAL_TRAJECTORY_CMD STATE_SIM_WAIT_FOR_DELAYED_TRAJECTORY_CMD STATE_SIM_TRAJECTORY	STATE_SIM_WAIT_FOR_PUNCTUAL_FORCE_TORQUE_CMD STATE_SIM_WAIT_FOR_DELAYED_FORCE_TORQUE_CMD STATE_SIM_FORCE_TORQUE
RSP_SVR_OUT_TARGET_CURR_ATT_U	N/M		$(q_{curr,t})^U$		
RSP_SVR_OUT_TARGET_CURR_RATE_U	N/M		$\left(\omega^{Body}_{curr,t}\right)^U$		
RSP_SVR_OUT_CHASER_REQ_POS_CLW	N/M		$r^{CLW}_{req,c}$		N/M
RSP_SVR_OUT_CHASER_REQ_SPEED_CLW	N/M		$\dot{r}^{CLW}_{req,c}$		N/M
RSP_SVR_OUT_CHASER_REQ_ATT_CLW	N/M		$(q_{req,c})^{CLW}$		N/M
RSP_SVR_OUT_CHASER_REQ_RATE_CLW	N/M		$\left(\omega^{Body}_{req,c}\right)^{CLW}$		N/M
RSP_SVR_OUT_TARGET_REQ_POS_CLW	N/M		$r^{CLW}_{req,t}$		N/M
RSP_SVR_OUT_TARGET_REQ_SPEED_CLW	N/M		$\dot{r}^{CLW}_{req,t}$		N/M
RSP_SVR_OUT_TARGET_REQ_ATT_CLW	N/M		$(q_{req,t})^{CLW}$		N/M
RSP_SVR_OUT_TARGET_REQ_RATE_CLW	N/M		$\left(\omega^{Body}_{req,t}\right)^{CLW}$		N/M
RSP_SVR_OUT_CHASER_REQ_POS_U	N/M		$r^{U}_{req,c}$		N/M
RSP_SVR_OUT_CHASER_REQ_SPEED_U	N/M		$\dot{r}^{U}_{req,c}$		N/M
RSP_SVR_OUT_CHASER_REQ_ATT_U	N/M		$(q_{req,c})^{U}$		N/M
RSP_SVR_OUT_CHASER_REQ_RATE_U	N/M		$\left(\omega^{Body}_{req,c}\right)^{U}$		N/M
RSP_SVR_OUT_TARGET_REQ_POS_U	N/M		$r^{CLW}_{req,t}$		N/M
RSP_SVR_OUT_TARGET_REQ_SPEED_U	N/M		$\dot{r}^{U}_{req,t}$		N/M
RSP_SVR_OUT_TARGET_REQ_ATT_U	N/M		$(q_{req,t})^{U}$		N/M
RSP_SVR_OUT_TARGET_REQ_RATE_U	N/M		$\left(\omega^{Body}_{req,t}\right)^{U}$		N/M
RSP_SVR_OUT_CHASER_CMD_POS	$r^{CLW}_{finit,c}$		$r^{CLW}_{cnd,c}$		
RSP_SVR_OUT_CHASER_CMD_ATT	$\left(q_{finit,c}\right)^{CLW}$		$(q_{cmd,c})^{CLW}$		

Continued on next page

Table 2.13 – *Continued from previous page*

	STATE_FACILITY_INIT	STATE_SAFE_SLOW_DOWN STATE_WAIT_FOR_CONNECTION STATE_WAIT_FOR_INIT_CMD	STATE_INIT_WAIT_FOR_PUNCTUAL_CMD STATE_INIT	STATE_SIM_WAIT_FOR_PUNCTUAL_TRAJECTORY_CMD STATE_SIM_WAIT_FOR_DELAYED_TRAJECTORY_CMD STATE_SIM_TRAJECTORY	STATE_SIM_WAIT_FOR_PUNCTUAL_FORCE_TORQUE_CMD STATE_SIM_WAIT_FOR_DELAYED_FORCE_TORQUE_CMD STATE_SIM_FORCE_TORQUE
RSP_SVR_OUT_TARGET_CMD_POS	$\boldsymbol{r}_{finit,t}^{CLW}$	$\boldsymbol{r}_{cmd,t}^{CLW}$			
RSP_SVR_OUT_TARGET_CMD_ATT	$\left(q_{finit,t}\right)^{CLW}$	$(q_{cmd,t})^{CLW}$			

Current chaser and target states in CLW coordinates have no meaning during STATE_FACILITY_INIT. Position and attitude are taken directly as the return values of the EPOS CMD interface which gives valid values only if the facility has been successfully initialized (Move to Start finished, compare Sec.1.2.4). Speed and angular velocity are calculated from position and attitude.

Current spacecraft states expressed in the User frame are converted from its counterparts in CLW coordinates. This coordinate transformation can be carried out not before a proper CMD_INIT packet has been received which determines the type of coordinate transformation (RSP_CLI_PARAM_COSY, e.g. RSP_PACKET_COSY). Therefore, they have no meaning from STATE_FACILITY_INIT to STATE_WAIT_FOR_INIT_CMD.

Requested robot states in U frame equal the commands received from RSP client. Its transformed version in CLW coordinates is used by RSP server to determine the interpolated trajectory. Both have meaning only during trajectory commanding (including the initial trajectory). During commanding of force and torque, the integrator embedded in the EPOS ACS/RT Simulink model provides robot states for each time step (possibly to be transformed into CLW coordinates) and there are no requested states sent by RSP client. Due to that reason, requested robot states have no meaning during force/torque commanding.

Commanded robot positions and attitudes are fed directly to the EPOS CMD interface. During STATE_FACILITY_INIT EPOS uses these values for "Move to Start". Therefore, in this state, the outputs are set to the according initial values, specified as RSP server parameters RSP_SVR_PARAM_CHASER_FACILITY_INIT_POS, RSP_SVR_PARAM_CHASER_FACILITY_INIT_ATT, RSP_SVR_PARAM_TARGET_FACILITY_INIT_POS and RSP_SVR_PARAM_TARGET_FACILITY_INIT_ATT. In all other states, these outputs provide the robot commands requested by RSP client and interpolated by RSP server, or given by the integrator outputs.

Tab.2.14 summerizes all RSP server output values which are connected to the integrator embedded in the ACS/RT Simulink model. The integrator enable signal RSP_SVR_OUT_INT_ENABLE indicates at which point the integrator is to start working. This point in time is reached as soon as the initial trajectory is finished. Hence, it is 0 in all states but those active during force/torque commanding, e.g. STATE_SIM_WAIT_FOR_PUNCTUAL_FORCE_TORQUE_CMD, STATE_SIM_FORCE_TORQUE and STATE_SIM_WAIT_FOR_DELAYED_FORCE_TORQUE_CMD.

In a similar fashion, force and torque for chaser and target have meaning only during regular force/torque simulation, where they are given by the received RSP packets force and torque fields. Up to STATE_INIT, provided that the initial time step has not been reached yet, the outputs have no meaning. As soon as the initial time step is reached in state STATE_INIT, the outputs are set to the

RSP_SVR_OUT_STATE	STATE_FACILITY_INIT STATE_SAFE_SLOW_DOWN STATE_WAIT_FOR_CONNECTION STATE_WAIT_FOR_INIT_CMD	STATE_INIT next hit not yet reached STATE_INIT_WAIT_FOR_PUNCTUAL_CMD	STATE_INIT next hit reached, MODE_TRAJECTORY	STATE_INIT next hit reached, MODE_FORCE_TORQUE	STATE_SIM_WAIT_FOR_PUNCTUAL_TRAJECTORY_CMD STATE_SIM_WAIT_FOR_DELAYED_TRAJECTORY_CMD STATE_SIM_TRAJECTORY	STATE_SIM_WAIT_FOR_PUNCTUAL_FORCE_TORQUE_CMD STATE_SIM_WAIT_FOR_DELAYED_FORCE_TORQUE_CMD STATE_SIM_FORCE_TORQUE
RSP_SVR_OUT_CHASER_INT_INIT_POS_U	N/M	$\boldsymbol{r}^{U}_{init,c}$	N/M	$\boldsymbol{r}^{U}_{init,c}$	N/M	$\boldsymbol{r}^{U}_{init,c}$
RSP_SVR_OUT_CHASER_INT_INIT_SPEED_U	N/M	$\dot{\boldsymbol{r}}^{U}_{init,c}$	N/M	$\dot{\boldsymbol{r}}^{U}_{init,c}$	N/M	$\dot{\boldsymbol{r}}^{U}_{init,c}$
RSP_SVR_OUT_CHASER_INT_INIT_ATT_U	N/M	$(q_{init,c})^{U}$	N/M	$(q_{init,c})^{U}$	N/M	$(q_{init,c})^{U}$
RSP_SVR_OUT_CHASER_INT_INIT_RATE_U	N/M	$\left(\boldsymbol{\omega}^{Body}_{init,c}\right)^{U}$	N/M	$\left(\boldsymbol{\omega}^{Body}_{init,c}\right)^{U}$	N/M	$\left(\boldsymbol{\omega}^{Body}_{init,c}\right)^{U}$
RSP_SVR_OUT_TARGET_INT_INIT_POS_U	N/M	$\boldsymbol{r}^{U}_{init,t}$	N/M	$\boldsymbol{r}^{U}_{init,t}$	N/M	$\boldsymbol{r}^{U}_{init,t}$
RSP_SVR_OUT_TARGET_INT_INIT_SPEED_U	N/M	$\dot{\boldsymbol{r}}^{U}_{init,t}$	N/M	$\dot{\boldsymbol{r}}^{U}_{init,t}$	N/M	$\dot{\boldsymbol{r}}^{U}_{init,t}$
RSP_SVR_OUT_TARGET_INT_INIT_ATT_U	N/M	$(q_{init,t})^{U}$	N/M	$(q_{init,t})^{U}$	N/M	$(q_{init,t})^{U}$
RSP_SVR_OUT_TARGET_INT_INIT_RATE_U	N/M	$\left(\boldsymbol{\omega}^{Body}_{init,t}\right)^{U}$	N/M	$\left(\boldsymbol{\omega}^{Body}_{init,t}\right)^{U}$	N/M	$\left(\boldsymbol{\omega}^{Body}_{init,t}\right)^{U}$
RSP_SVR_OUT_CHASER_INT_FORCE	N/M	N/M	N/M	RSP_PACKET_CHASER_FORCE	N/M	RSP_PACKET_CHASER_FORCE
RSP_SVR_OUT_CHASER_INT_TORQUE	N/M	N/M	N/M	RSP_PACKET_CHASER_-TORQUE	N/M	RSP_PACKET_CHASER_TORQUE
RSP_SVR_OUT_TARGET_INT_FORCE	N/M	N/M	N/M	RSP_PACKET_TARGET_FORCE	N/M	RSP_PACKET_TARGET_FORCE
RSP_SVR_OUT_TARGET_INT_TORQUE	N/M	N/M	N/M	RSP_PACKET_TARGET_-TORQUE	N/M	RSP_PACKET_TARGET_TORQUE
RSP_SVR_OUT_INT_ENABLE	0	0	0	0	0	1

Table 2.14.: Output values of RSP Server for the Integrator depending on Server State, Mode and Time Step. N/M - No Meaning. All RSP packet fields given here correspond to the received packet from the RemoteSim-Client, e.g. RSP_SVR_IN_RECVD_RSP_PACKET.

force/torque values. This is repeated on a regular basis as illustrated in Sec.2.6.2 during force/torque commanding. Logically, the outputs have no meaning at all with MODE_TRAJECTORY.

The integrator needs initial values from where to start integration. These initial values comprise another set of RSP server outputs listed in Tab.2.14. The initial values are transmitted in the CMD_INIT packet and equal the states the initial trajectory is to realize during STATE_INIT_WAIT_FOR_PUNCTUAL_CMD and STATE_INIT. The initial values must be available to the integrator in the time step RSP_SVR_OUT_INT_ENABLE is 1 for the first time in a simulation run. However, for simplicity, the outputs are set to the initial values as soon as they are received in the CMD_INIT packet, e.g. starting with STATE_INIT_WAIT_FOR_PUNCTUAL_CMD. The integrator "reacts" on the initial values not before the enable signal is different from 0, anyway.

Tab.2.15 lists other RSP server outputs which have not been covered in the tables above. Dependence

Table 2.15.: Values of RSP server outputs (client state, mode, cosy, error). RSP_PACKET_STATE referes to RSP_SVR_IN_RECVD_RSP_PACKET

RSP_SVR_OUT_CLIENT_STATE	RSP_PACKET_STATE
RSP_SVR_OUT_MODE	as initialized by CMD_INIT packet, MODE_NOT_A_MODE before initialization
RSP_SVR_OUT_COSY	as initialized by CMD_INIT packet, COSY_NOT_A_COSY before initialization
RSP_SVR_OUT_ERROR	current server error

on server state is minor and therefore no distinction is made in Tab.2.15. RSP_SVR_OUT_CLIENT_STATE directly represents the RSP_PACKET_STATE field of the RSP packet received from the client. RSP_SVR_OUT_MODE and RSP_SVR_OUT_COSY inform about the current setup of the simulation. And finally, RSP_SVR_OUT_ERROR indicates whether an error occured. There are two possible values: ERROR_NOT_A_ERROR and ERROR_CMD_TO_LATE.

2.6.5. Detailed Client-Server Interaction

Sec.2.6.2 gave an overview of RSP client-server interaction in order to illustrate some principles as a basis for discussion of client and server behaviour. In this section the interaction is presented in more detail. Several UML timing diagrams show different stages of the remote simulation process. These timing diagrams depict concrete examples. Time steps may not represent realistic values or ranges of time. Rather, they are chosen such that the essential relations and ideas are conveyed clearly. Note also that both RSP server and RSP client send a RSP packet at each time step (an intact connection provided). In some cases, this may lead to multiple transmissions of identical packets, of which only the first one changed is considered. Of course, not all of these packets can be depicted. Only the relevant packets are shown as timing diagram messages where they support understanding. In general, RSP client always is informed about the RSP server's state, since the packet sent by the server at each time step contains this information. In the timing diagrams, client and server states are given as numbers which are associated to its textual expressions in separate comment boxes. Note that the numbers refer to the constants as actually implemented but can be considered to be arbitrary. There is no reason why state A may be represented by 3 and state B by 77. Client-server interaction is treated in a somewhat chronological order, from establishing a new connection to regular simulation. Both trajectory and force/torque commanding is illustrated. Moreover, it is shown how RSP handles delayed CMDs (i.e. due to a perturbed ethernet connection).

2.6.5.1. Establishing a new Connection

Facility initialization and etablishing a new RSP connection is carried out by RSP server and client widely independently from each other. Fig.2.15 shows an example. In the upper half of the timing

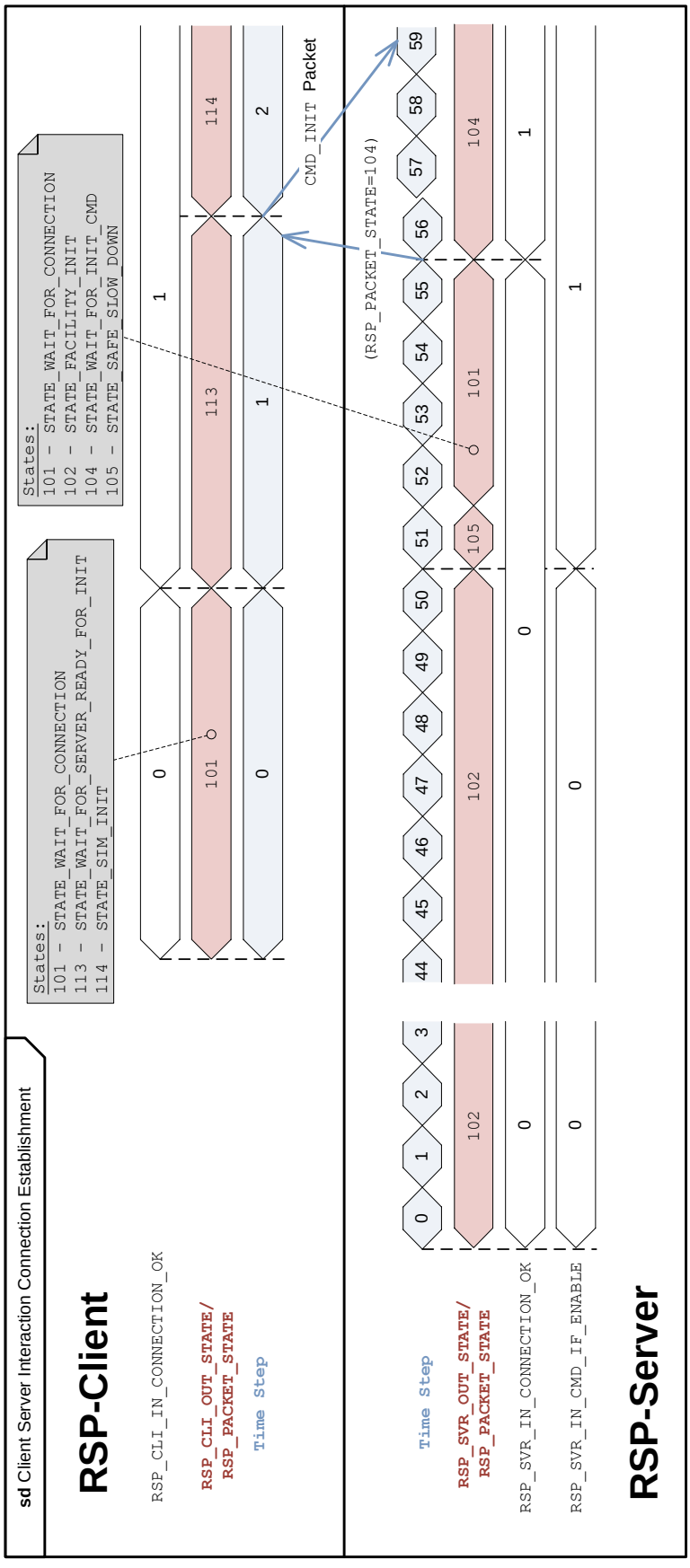

Figure 2.15.: UML Timing Diagram of Client-Server Interaction during Establishment of a new Connection

diagram, client time step, state and the connection status (signalled by SCP) are presented. In the lower half, time step of server as well as server state, connection status and the enable signal coming from the EPOS CMD interface are given.

In this example, RSP server (e.g. ACS/RT Simulink model) is started first. It stays in STATE_FACILITY_INIT until the facility indicates that it is ready for synchronous commanding by a 1 at the RSP server input RSP_SVR_IN_CMD_IF_ENABLE. The state that follows is STATE_SAFE_SLOW_DOWN. Its purpose is to slow the robots down to zero speed (keep station) and is used not only at this point but also if the remote simulation is stopped. Here, the "Move to Start" process has already stopped the robots and hence this state takes only a single time step before RSP server transits to STATE_WAIT_FOR_CONNECTION. To allow the server to leave this state, SCP has to signal an intact connection to a RSP client (RSP_SVR_IN_CONNECTION_OK). Then RSP server is ready for initializing a remote simulation, which is indicated by STATE_WAIT_FOR_INIT_CMD. Up till now, there has been no interaction between server and client.

RSP client is started later in example depicted in Fig.2.15. It starts in STATE_WAIT_FOR_CONNECTION. As soon as SCP signals a valid connection to a RSP server, STATE_WAIT_FOR_SERVER_READY_FOR_INIT. RSP client stays in this state until the RSP_PACKET_STATE field of a received packet (e.g. RSP_CLI_IN_RECVD_RSP_PACKET) contains the server state STATE_WAIT_FOR_INIT_CMD. This is the first point in time when RSP server and client start interacting. As soon as the client is informed about the server being ready for initializing a remote simulation, it sends a proper CMD_INIT packet and transits to STATE_SIM_INIT.

These preliminary actions are required for trajectory and for force/torque commanding. Both will be treated in the subsequent sections.

2.6.5.2. Trajectory Commanding

Fig.2.16 illustrates regular trajectory commanding as an UML timing diagram. Regular means that CMDs are received in time and no extrapolation is necessary. Fig.2.16 is similar to the example of Sec.2.6.2. Therefore, RSP principles like hits and forerun are not repeated here. Rather, emphasis lies and state transitions, key outputs and the interaction between RSP server and client.

In addition to time step and state, Fig.2.16 also shows the simulation enable output of RSP client and mode, cosy and integrator enable outputs of RSP server. At the client side, state transitions are comparatively simple. In parallel with transmission of the CMD_INIT packet, the client changes to STATE_SIM_INIT and stays in this state until timing (calculated before) asks for transmission of the first CMD_TRAJECTORY packet. Then, in STATE_SIM_TRAJECTORY such a packet is sent at each time step. In fact, interaction is quite one sided. The only point, when RSP really needs information from the server is at the end of Fig.2.15, e.g. at the beginning of Fig.2.16: RSP client waits until RSP server is ready for commanding (including utilization of server sample time in RSP_CLI_IN_RECVD_RSP_PACKET at this stage). From there, the client merely informs the server about initial time step and (implicitly via sample times) about interpolation intervals in the CMD_INIT packet. It then sends trajectory CMDs according to this initial time step and the interpolation intervals without any further "consultation" with RSP server. One of the most important outputs RSP client provides is the simulation enable signal. It is used to control the execution of the user simulation model which is connected to RSP client in the remote Simulink simulation. Fig.2.16 clearly shows that RSP_CLI_OUT_SIM_ENABLE changes from 0 to 1 one step ahead of transmission of the first trajectory CMD. One would expect this to happen precisely when the first spacecraft states are needed for the first CMD. And this is very true. However, in a Simulink model, connecting the enable signal to a enabled subsystem which has inputs being connected directly to a block containing the RSP client would result in an algebraic loop. (Simulink cannot calculate a block's output before knowing its input.) A unit delay must be inserted. Since the sample time of the remote simulation is likely to be relatively large (in the order of $100ms$ at least), this might introduce a considerable error. Activating the enable signal one step earlier eliminates this error.

After having received the CMD_INIT packet, RSP server calculates the initial trajectory and has the robots move accordingly. The server waits for the first trajectory CMD which is indicated by STATE_INIT_WAIT_FOR_PUNCTUAL_CMD. Moreover, the CMD_INIT packet already determines type of commanding and type of coordinate transformation. Therefore, the output RSP_SVR_OUT_MODE changes to MODE_TRAJECTORY and RSP_SVR_OUT_COSY to COSY_FEEDTHROUGH or COSY_ECI_2_CLW according to the value in the CMD_INIT packet. The integrator enable output is also shown in Fig.2.16 since it

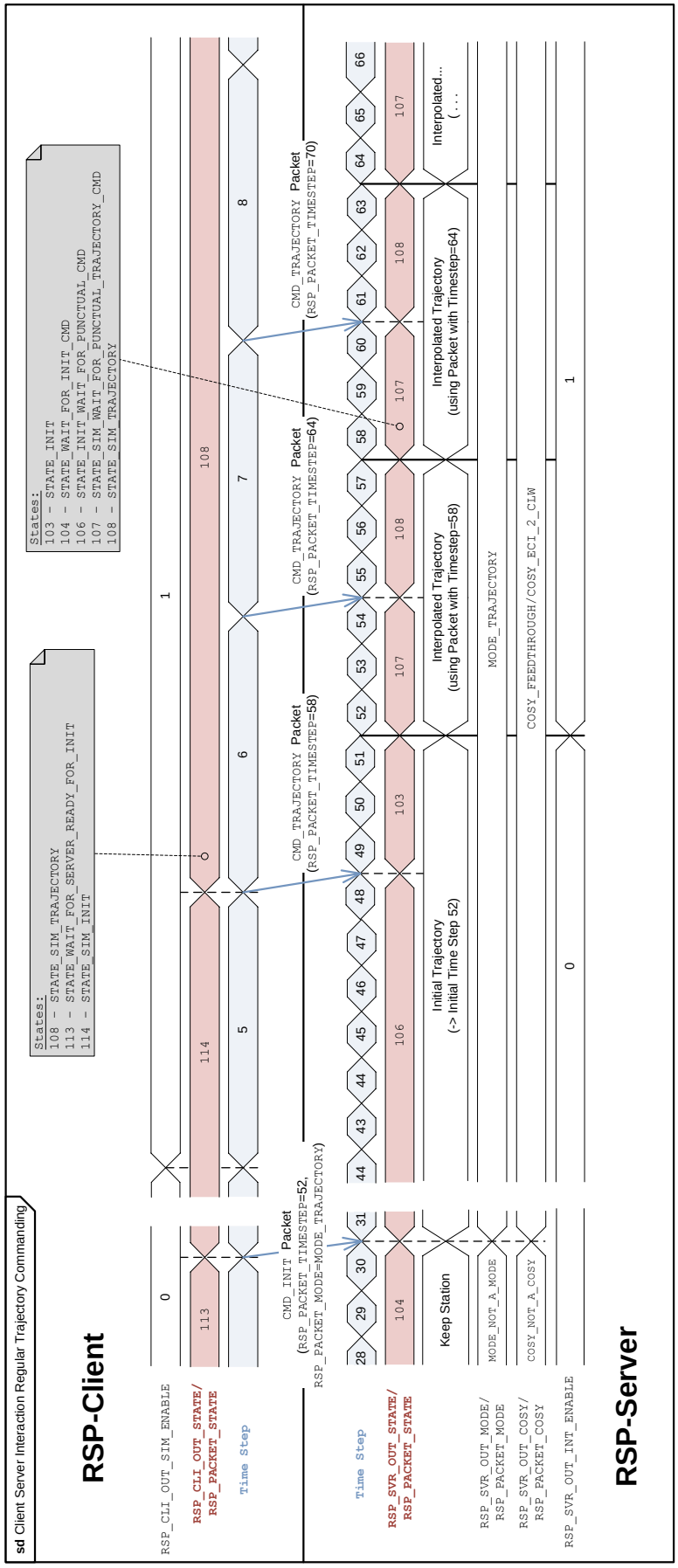

Figure 2.16.: UML Timing Diagram of Client-Server Interaction during regular Trajectory Commanding

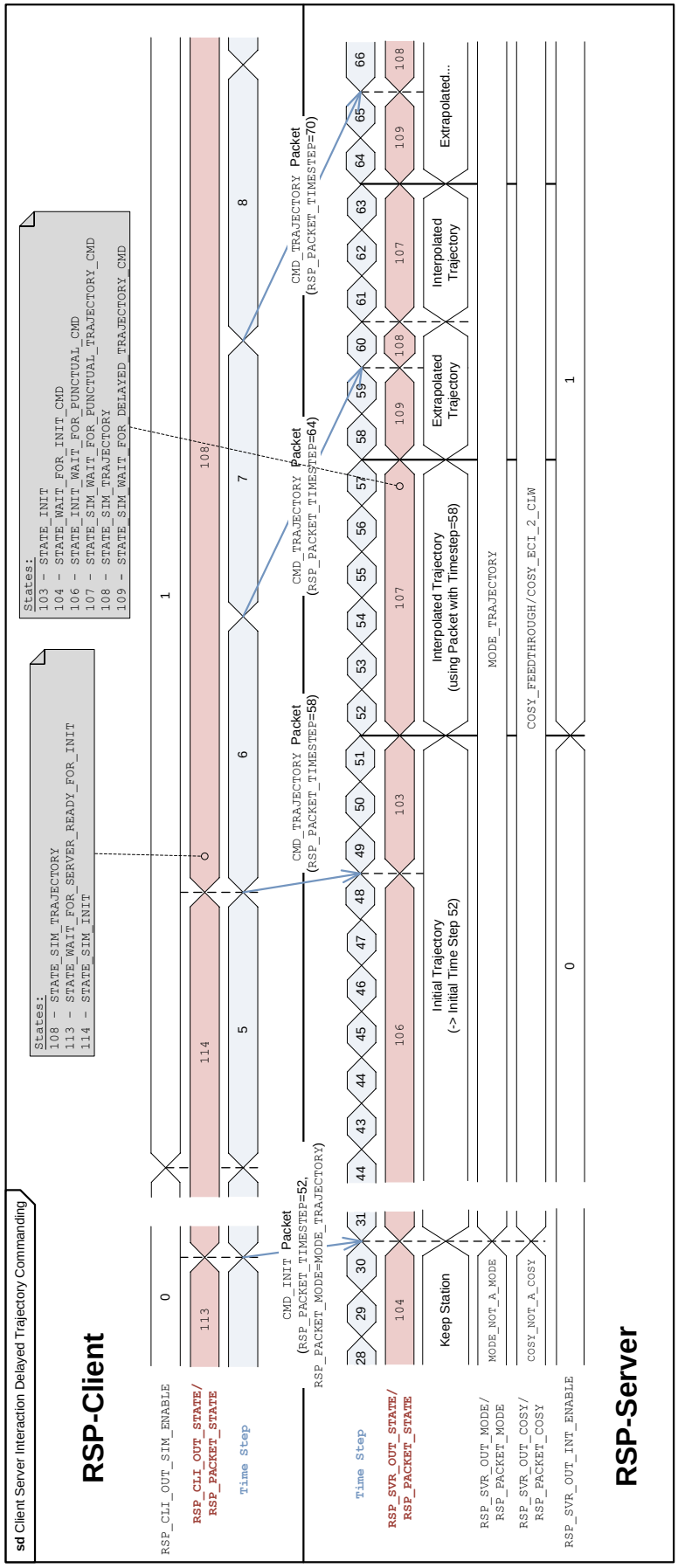

Figure 2.17.: UML Timing Diagram of Client-Server Interaction during delayed Trajectory Commanding

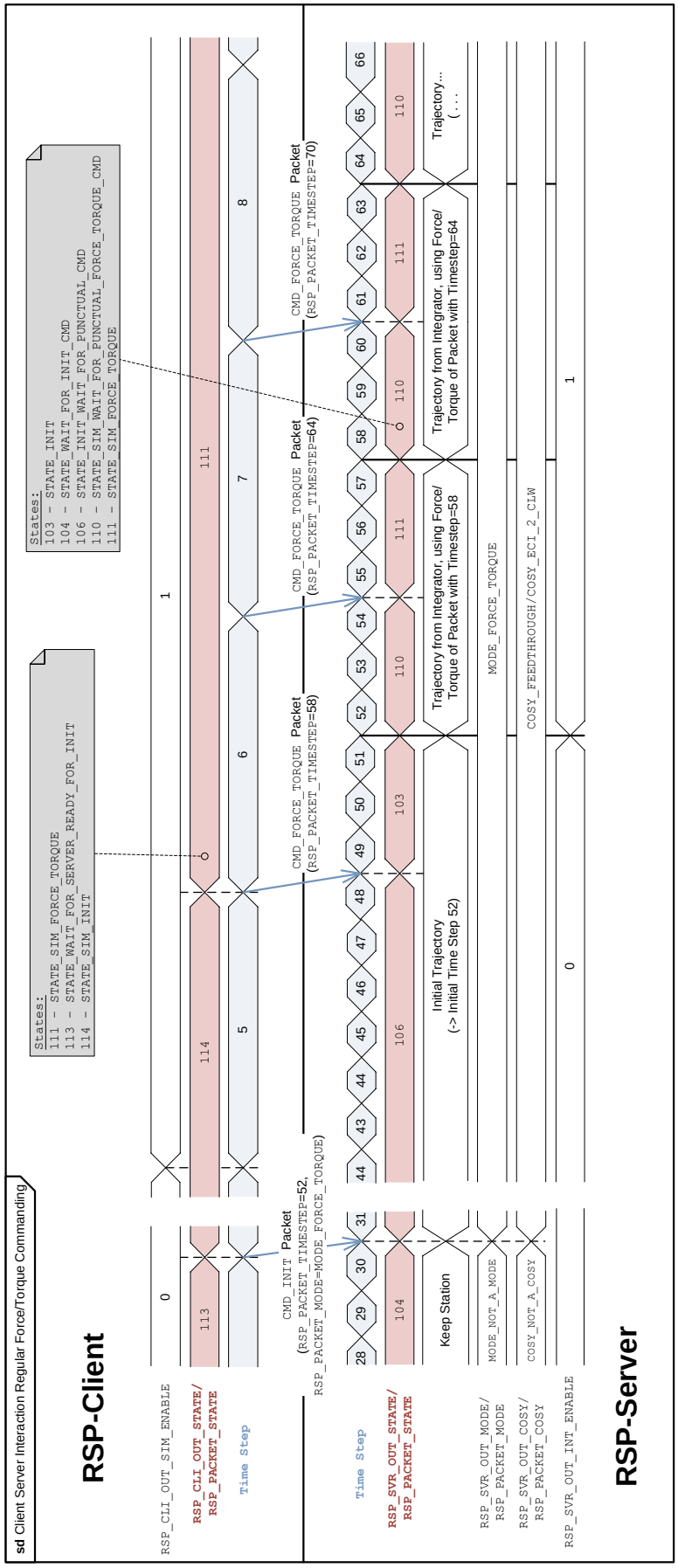

Figure 2.18.: UML Timing Diagram of Client-Server Interaction during regular Force/Torque Commanding

is activated during trajectory commanding also, for simplicity. It will be discussed with force/torque commanding below. As soon as RSP client sends the first trajectory CMD, the server transits to STATE_INIT and stays there until the initial trajectory is finished. Thereupon RSP server oscillates between STATE_SIM_WAIT_FOR_PUNCTUAL_TRAJECTORY_CMD and STATE_SIM_TRAJECTORY. It waits for the next trajectory CMD, saves the requested states upon receipt, waits until the next hit, calculates the interpolated trajectory and waits again for the next trajectory CMD...

This loop is carried out as long as trajectory CMDs are received in time. Now suppose that the ethernet is temporarily perturbed such that packets take an unusual long time from RSP client to RSP server. Fig.2.17 is similar to Fig.2.16 except that, beginning from the second trajectory packet, CMDs are received delayed. Now in STATE_SIM_WAIT_FOR_PUNCTUAL_TRAJECTORY_CMD no proper trajectory CMD is received, RSP server assumes that such a packet will come late and transits to STATE_SIM_WAIT_FOR_DELAYED_TRAJECTORY_CMD when next hit is reached. During regular commanding, the robots would follow the new interpolated trajectory. Here, however, the old interpolation is used to extrapolate the robots' states until a delayed packet is received. Upon receipt of this delayed CMD packet RSP server changes to STATE_SIM_TRAJECTORY just as during regular simulation. Since next hit has already been reached a few time steps before, STATE_SIM_TRAJECTORY is active for only one time step, the new interpolated trajectory is calculated for the rest of the present interpolation interval and RSP server transits to STATE_SIM_WAIT_FOR_PUNCTUAL_TRAJECTORY_CMD awaiting the next CMD_TRAJECTORY packet. If this packet is also late, as depicted in Fig.2.17, the process begins anew.

2.6.5.3. Force/Torque Commanding

The RSP server state machine already showed that force/torque commanding is quite similar to trajectory commanding. This is confirmed looking at Fig.2.18. The states STATE_SIM_WAIT_FOR_PUNCTUAL_TRAJECTORY_CMD and CMD_TRAJECTORY are replaced by STATE_SIM_WAIT_FOR_PUNCTUAL_FORCE_TORQUE_CMD and CMD_FORCE_TORQUE. RSP_SVR_OUT_MODE now is MODE_FORCE_TORQUE instead of MODE_TRAJECTORY. But the most important difference is the fact that no interpolation is required. The robot states to be commanded are calculated by the integrator at each server time step. The interpolation interval can be considered to be replaced by a "force and torque" interval. The requested force and torque (received in a CMD_FORCE_TORQUE packet) is fed to the integrator at each hit and stays the same between next hit and next-but-one hit just as an interpolated trajectory is valid until it is replaced in the next interpolation interval.

The output RSP_SVR_OUT_INT_ENABLE plays an important role. It is an enable signal which controls the integrator of the EPOS ACS/RT Simulink model, analogous to the user model of the remote simulation. Integration is to start, when this signal changes from 0 to 1. This occurs precisely when RSP server changes its state from STATE_INIT to STATE_SIM_WAIT_FOR_PUNCTUAL_FORCE_TORQUE_CMD. It may be necessary to include a unit delay in a simulation using this signal in order to avoid an algebraic loop. In contrast to the simulation enable signal of RSP client, no attempts are made to counteract this delay for the sake of simplicity. Since server sample time is small, e.g. sample frequency is high, such a unit delay for the integrator enable signal won't have any noteworthy impact.

3. Remote Control Implementation

3.1. Excursion: Matlab/Simulink c-mex S-Functions

RemoteSim-Client and RemoteSim-Server are implemented as c-mex S-Functions. This section gives a very brief description of this kind of custom Simulink block, based on [17], as background for the subsequent sections in this chapter.

A c-mex S-Function is a Simulink block which is described in behaviour by C or C++ code. This code must comply with the S-Function API. It is comprised of a set of callback routines allowing to interact with the Simulink engine. Some examples: The callback `mdlInitializeSizes` is executed by Simulink to determine the number of inputs, outputs and parameters as well as several of its attributes. `mdlStart` is called just before a Simulink model is started. In contrast, `mdlOutputs` is executed at each time step to calculate the block's outputs. And `mdlTerminate` is called at the end of a simulation. These are just some of all callback routines. Only few of them are obligatory. The others are optional. A minimum c-mex S-Function is a c-source file containing the obligatory S-Function callback routines and including mex and Simulink specific header files and an associated S-Function block with the compiled source file selected. This master source file can be complemented by other modules. Especially, the wrapper concept illustrated in [17] is an interesting possibility of integrating other modules and is used for RemoteSim. The callback routines in the master source file call wrapper functions defined in a separate file. This wrapper function then executes other code which again is located in other files. The advantage is that the actual code can be kept independent from the S-Function and could be used in other projects separately (maybe as a stand-alone application). The wrapper serves as a bridge between implementation independent code and S-Function API. The definition of S-Function block masks allows to enter parameters in a comfortable fashion. With restrictions, S-Functions can be used with Simulink Real-Time Workshop. An additional file, a so-called TLC-file (Target Language Compiler) is required to provide Simulink with information about how to integrate the S-Function code with the code generated by Real-Time Workshop.

3.2. Coding Convention

RemoteSim is implemented in C/C++. The author wrote the code widely compliant with usual coding convention. Classes are written with leading capital letters, no space between words and with a C as first letter. Example: `CRemoteSimClient`. Enumerations and structures are written similarly but with a leading E or S respectively. Example: `EMode`, `SFormtationState`. Member variables have a leading m, are written with lower-case characters where different words are separated by underlines. Example: `m_connection_broken`. Class methods are written without underlines between consecutive words. The first letter in each word is a lower-case character, except of the first word. Example: `runThisTimestep`. Any type of constants, enumerations and preprocessor defines, are written in capital letters with underlines between consecutive words. Example: `MODE_TRAJECTORY`.

3.3. RemoteSim-Client Block

3.3.1. Block Inputs, Outputs and Parameters

Fig.3.1 shows the Simulink Block of RemoteSim-Client. Note that RemoteSim-Client incorporates both RSP component and SCP component for realizing the complete client.

The inputs of the block consist of requested chaser and target states for trajectory commanding as well as chaser and target force and torque for force/torque commanding. Usually, the user will connect the outputs of its simulation to these inputs. Tab.A.1 in the appendix summarizes the inputs with unit and format.

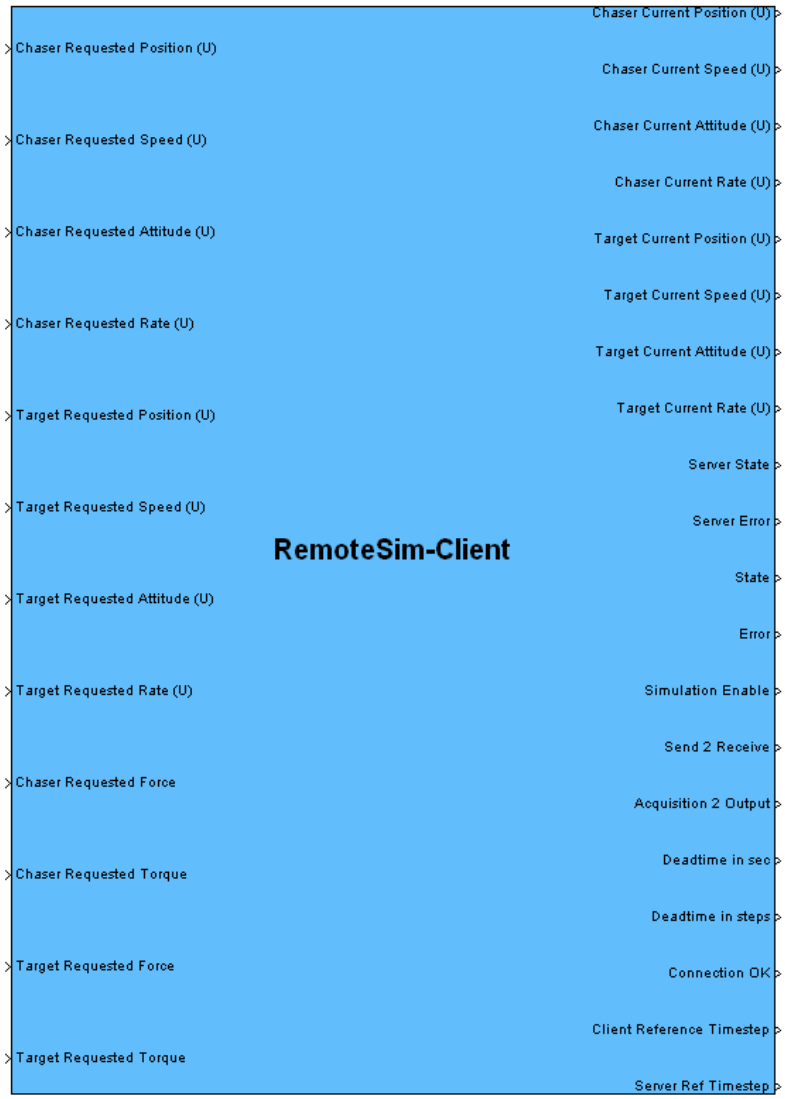

Figure 3.1.: Simulink Block of RemoteSim-Client

Block outputs comprise current chaser and target state, expressed in U frame, information about client state, server state and server error. There are also outputs corresponding to connection quality (Send 2 Receive, Acquisition 2 Output, Deadtime in sec and Deadtime in steps) and connection status. Currently determined reference time steps can be displayed. One of the outputs is the required simulation enable signal for controlling user simulation execution. Save for the enable signal, all outputs are for monitoring purposes only and are not obligatorily required. Tab.A.2 in the appendix summarizes the outputs with unit and format.

RemoteSim-Client block requires several parameters to be specified in the S-Function mask. There are parameters needed for the network connection: control and data port, IP address of RemoteSim-Server and a timeout for detecting a broken connection. Moreover, type of commanding and type of coordinate transformation have to be chosen by the user. The desired forerun as well as the initial timespan, e.g. the time the robots take for reaching the initial conditions, must be set. Finally, the sample time which must equal the simulation's sample time is required. Tab.A.3 in the appendix summarizes the parameters with unit, format and example. The examples are reasonable values for the EPOS configuration at the time this thesis was created.

The header file `remotesim_client_sfun_spec.h` defines input ports, output ports and parameters. See appendix E. Note that "SC1" refers to chaser S/C and "SC2" refers to target S/C in the file.

3.3.2. Software Structure

The UML object digram depicted in Fig.3.2 gives an overview of RemoteSim-Client block software structure. Note that the object diagram illustrates the instances of C++ classes and their relations.

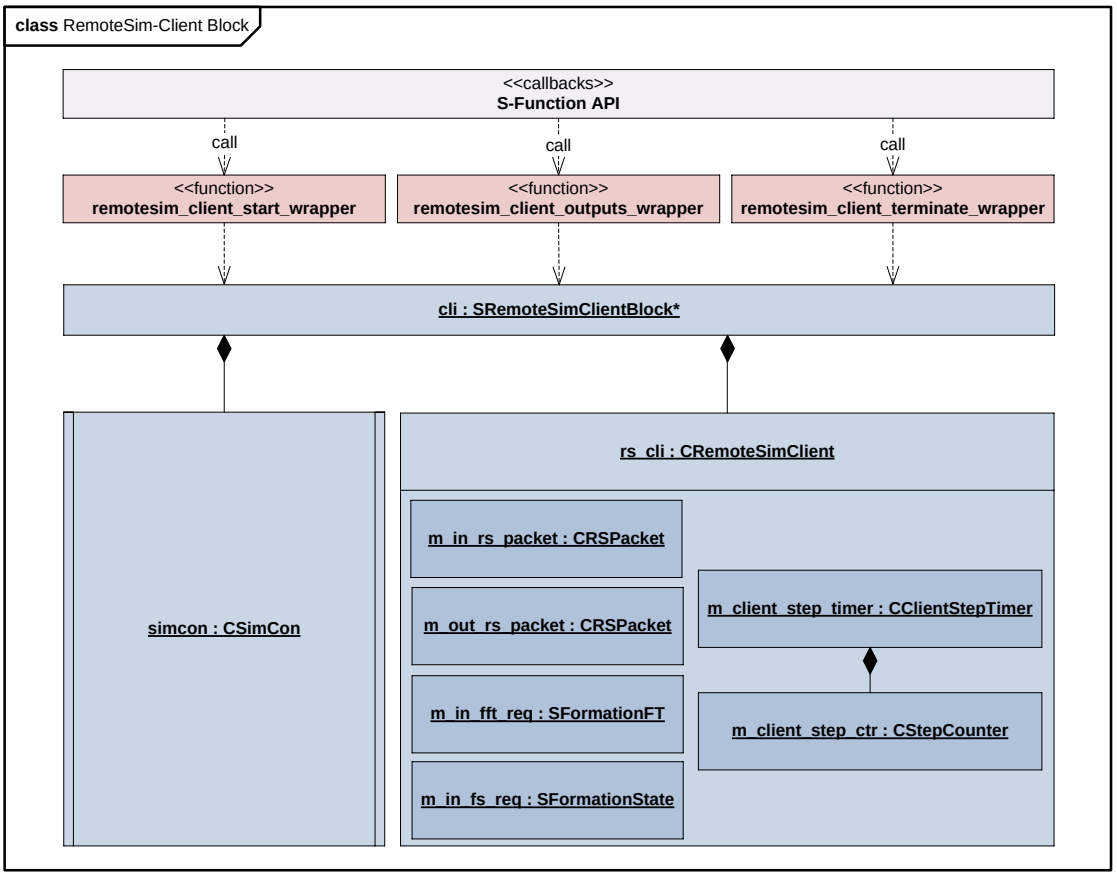

Figure 3.2.: UML Object Diagram of RemoteSim-Client Block Software Structure

Only the most important components are included insofar as vital for understanding, primitive member variables are not depicted.

The order from top to bottom can be considered to represent the hierarchical structure of the software. On the top level rests the S-Function API consisting of the various callback functions which are executed by the Simulink engine during simulation. The callback methods call the wrapper functions on their part. As shown, these wrapper functions constitute a bridge between the Simulink specific implementation and the widely independent part of the software. A `struct` called `SRemoteSimClientBlock` is instanciated, accessed and destroyed by the wrapper functions. This `struct` incorporates one instance of `CSimCon` (see Sec.C.30) realizing SCP and one instance of `CRemoteSimClient` (see Sec.C.25) realizing the client role of RSP. Both classes are combined in the `struct` to make interaction with the API callbacks easier. (Thus, only one RWork vector has to be allocated. The wrapper function can handle this single `void` pointer as a return value. See [17] for more details.) `CSimCon` is treated separately in Sec.3.5, since it is also part of RemoteSim-Server block and is too complicated to be depicted in Fig.3.2 also. The instance of `CRemoteSimClient` contains two instances of a RSP packet (`CRSPacket`, described in Sec.C.29), `m_in_rs_packet` and `m_out_rs_packet`. The former contains the data from SCP representing the RSP packet sent by RemoteSim-Server (`RSP_SVR_OUT_SEND_RSP_PACKET`, e.g. `RSP_CLI_IN_RECVD_RSP_PACKET`). The latter contains the data representing the RSP packet to be sent to RemoteSim-Server (`RSP_SVR_IN_RECVD_RSP_PACKET`, e.g. `RSP_CLI_OUT_SEND_RSP_PACKET`). The requested spacecraft states are buffered in an instance `m_in_fs_req` of the `struct SFormationState` (described in Sec.C.43). Respectively, requested force and torque are buffered in an instance `m_in_fft_req` of the `struct SFormationFT`

(described in Sec.C.44). A very important component is m_client_step_timer (instanciation of CClientStepTimer). It is responsible for calculating simulation timing (initial server time step) and for signalling that the initial trajectory has been completed. CClientStepTimer contains an instance of CStepCounter for logging time steps. The former is described in Sec.C.4 and the latter in Sec.C.36 in detail.

3.3.3. Working Principle

The function remotesim_client_start_wrapper is called by the S-Function API callback method mdlStart and instanciates cli. Moreover, it initializes simcon and rs_cli by its initialization methods CSimCon::init(...) and CRemoteSimClient::init(...) with RSP and SCP parameters (Tab.A.3).

The function remotesim_client_outputs_wrapper is called by the S-Function API callback method mdlOutputs at each time step. In Fig.3.3 the routine is described as an UML activity diagram. First, chaser and target states as well as force and torque are read from the inputs (req_fs_u : SFormationState and req_fft_u : SFormationFT) and are passed to the CRemoteSimClient instance. Moreover, reference time steps and connection status are read from simcon and handed to rs_cli. In terms of RSP and SCP inputs and outputs, this corresponds to the assignments

$$\begin{aligned} \texttt{RSP\_CLI\_IN\_CLIENT\_REF\_TIMESTEP} &= \texttt{SCP\_OUT\_HOME\_REF\_TIMESTEP} \\ \texttt{RSP\_CLI\_IN\_SERVER\_REF\_TIMESTEP} &= \texttt{SCP\_OUT\_TARGET\_REF\_TIMESTEP} \\ \texttt{RSP\_CLI\_IN\_CONNECTION\_OK} &= \texttt{SCP\_OUT\_CONNECTION\_OK} \end{aligned}$$

If a new RSP packet has been received by simcon, which is signalled by the boolean return value of the method CSimCon::isNew(), the packet is passed to the CRemoteSimClient instance.

$$\texttt{RSP\_CLI\_IN\_RECVD\_RSP\_PACKET} = \texttt{SCP\_OUT\_RECVD\_RSP\_PACKET}$$

If there is no new received RSP packet, this step is omitted. Next, this is followed by the methods CSimCon::beginThisTimestep() and CRemoteSimClient::runThisTimestep() wherein the actual processing takes place, i.e. running the RSP client state machine. Then, rs_cli provides a new RSP packet which is handed to simcon for transmission to RemoteSim-Server. In terms of Remote Simulation Protocol and Simulation Connection Protocol this yields

$$\texttt{SCP\_IN\_SEND\_RSP\_PACKET} = \texttt{RSP\_CLI\_OUT\_SEND\_RSP\_PACKET}$$

The wrapper function, e.g. the activity, is finished by getting all output values from simcon and rs_cli. Overall, remotesim_client_outputs_wrapper reads RemoteSim-Client inputs, supplies them to CSimCon and CRemoteSimClient instances, executes their main processing functions (which have to be called at each time step) and gets RemoteSim-Client outputs.

Finally, the function remotesim_client_terminate_wrapper is called by the S-Function API callback method mdlTerminate and destroys cli which implies closing the connection, deallocating memory etc..

To get a more complete picture of the processing at each time step, the public member functions CSimCon::beginThisTimestep() and CRemoteSimClient::runThisTimestep() are briefly described here. During executen of the former, the current time step is logged, the connection status checked and eventually repaired. There is no more processing in this function, since CSimCon includes a number of additional threads which do the work in the background, independently from simulation time steps (see Sec.3.5). The method CRemoteSimClient::runThisTimestep() is described in Fig.C.28. In summary, the client step timer is informed that a new time step has begun, the RSP client state machine is run (runStateMachine) and finally, all fields of m_out_rs_packet which are not set by the state machine individually are written to. See Sec.C.25 for a detailed description of CRemoteSimClient and its methods.

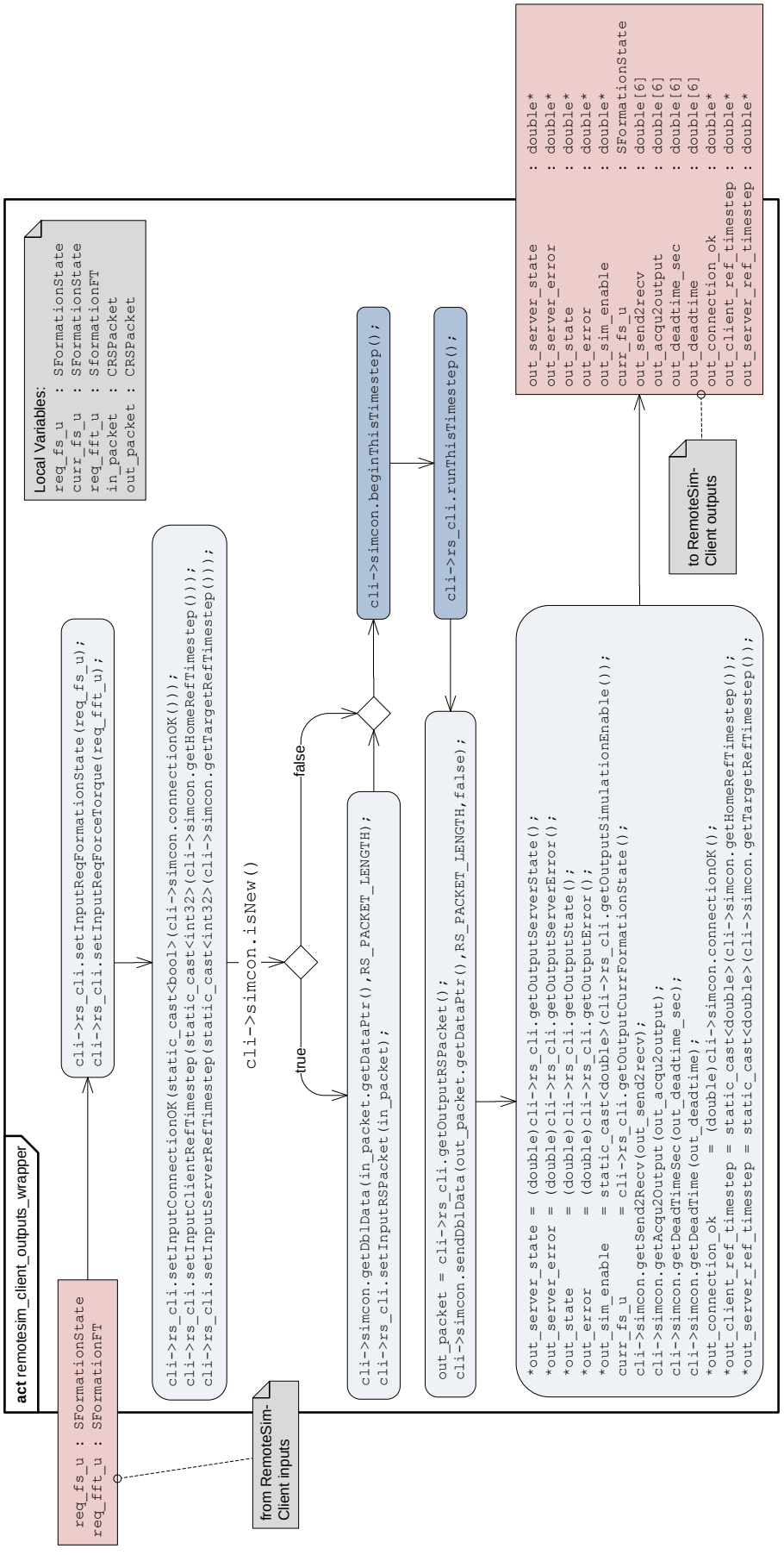

Figure 3.3.: UML Activity Diagram of the wrapper function `remotesim_client_-outputs_wrapper`

3.4. RemoteSim-Server Block

3.4.1. Block Inputs, Outputs and Parameters

Fig.3.4 shows the Simulink Block of RemoteSim-Server. Note that RemoteSim-Server incorporates both RSP component and SCP component for realizing the complete server.

Figure 3.4.: Simulink Block of RemoteSim-Server

RemoteSim-Server inputs include the enable signal as well as current chaser and target positions and attitudes coming from the EPOS CMD interface. Moreover, there are inputs for spacecraft states calculated by the integrator (only used with `MODE_FORCE_TORQUE`). And a reset input allows for resetting RemoteSim-Server after a finished remote simulation. Tab.B.1 in the appendix summarizes the inputs with unit and format.

RemoteSim-Server provides a large number of different outputs, most of which are for monitoring purposes. Commanded chaser and target position/attitude have to be connected to the EPOS CMD interface, integrator enable signal and integrator initial states are to be fed to the integrator inputs

as well as chaser and target force/torque. Thus, these outputs fullfill functional tasks. The following are for monitoring only. Information about server state, error, mode and cosy allow for determining current status of RemoteSim-Server. There are also outputs for current robot states, in U and CLW frame, and for requested robot states (received from the client) also in U and CLW frame. Similar to RemoteSim-Client, connection quality, connection status and reference time steps can be displayed. In addition, the actual forerun is provided and two signals indicate whether speed/acceleration limits are currently violated. Tab.B.2 in the appendix summarizes the outputs with unit and format.

All connection related RemoteSim-Server parameters are identical to those of RemoteSim-Client. Moreover, a speed limit, an angular speed limit, an acceleration limit and an angular acceleration limit can be specified to constrain robot movement accordingly. For both robots, a position and attitude must be provided for facility initialization. Braking acceleration and angular braking acceleration determines how fast the robots come to rest, if the remote simulation is stopped or connection brakes up. Finally, the simulation's sample time has to be provided. For the EPOS ACS/RT this usually is 0.004. Tab.B.3 in the appendix summarizes the parameters with unit, format and example. The examples are reasonable values for the EPOS configuration at the time this thesis was created.

The header file `remotesim_server_sfun_spec.h` defines input ports, output ports and parameters. See appendix E. Note that "SC1" refers to chaser S/C and "SC2" refers to target S/C in the file.

3.4.2. Software Structure

On the top-level, the structure of RemoteSim-Server software is quite similar to the one of RemoteSim-Client software, as can be deduced from Fig.3.5. There are three wrapper functions which are exe-

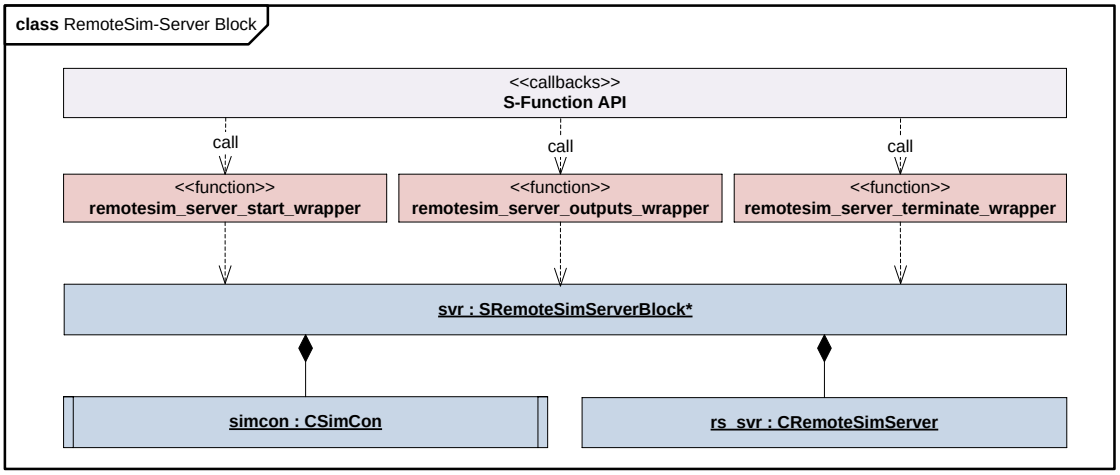

Figure 3.5.: UML Object Diagram of RemoteSim-Server Block Software Structure

cuted by the S-Function API callback methods. The `struct SRemoteSimServerBlock` is comprised of one instance of `CSimCon` and one of `CRemoteSimServer` (see Sec.C.26). The wrapper function `remotesim_server_start_wrapper` instanciates `svr`, `remotesim_server_outputs_wrapper` is executed at each time step and `remotesim_server_terminate_wrapper` destroys `svr` again. For better lucidity, the structure of `CRemoteSimServer` is described in a separate activity diagram, depicted in Fig.3.6.

There are a number of `SFormationStates` and `SFormationFTs` for buffering chaser and target states and force/torque, i.e. current states provided by the EPOS CMD interface or the next states to be commanded that have been received in a RSP packet. `m_step_timer` is an instance of `CStepTimer`. It is the equivalence of RemoteSim-Client's `CClientStepTimer`. However, its functionality is considerably larger. Among other tasks, it signals that a new interpolation interval is due, handles delayed CMDs and calculates the actual forerun upon receipt of a CMD packet. `CStepTimer` is described in Sec.C.38 in detail. `CRemoteSimServer` also includes an instance of `CCosyConverter` (see Sec.C.7) which handles the coordinate conversion outlined in Sec.2.6.4.6. It contains several member variables, vectors `CVec3D` (see Sec.C.42) and a quaternion `CQuaternion` (see Sec.C.22), needed for calculation. (Some values must be saved during ECI 2 CLW conversion, in order to realize the associated CLW 2

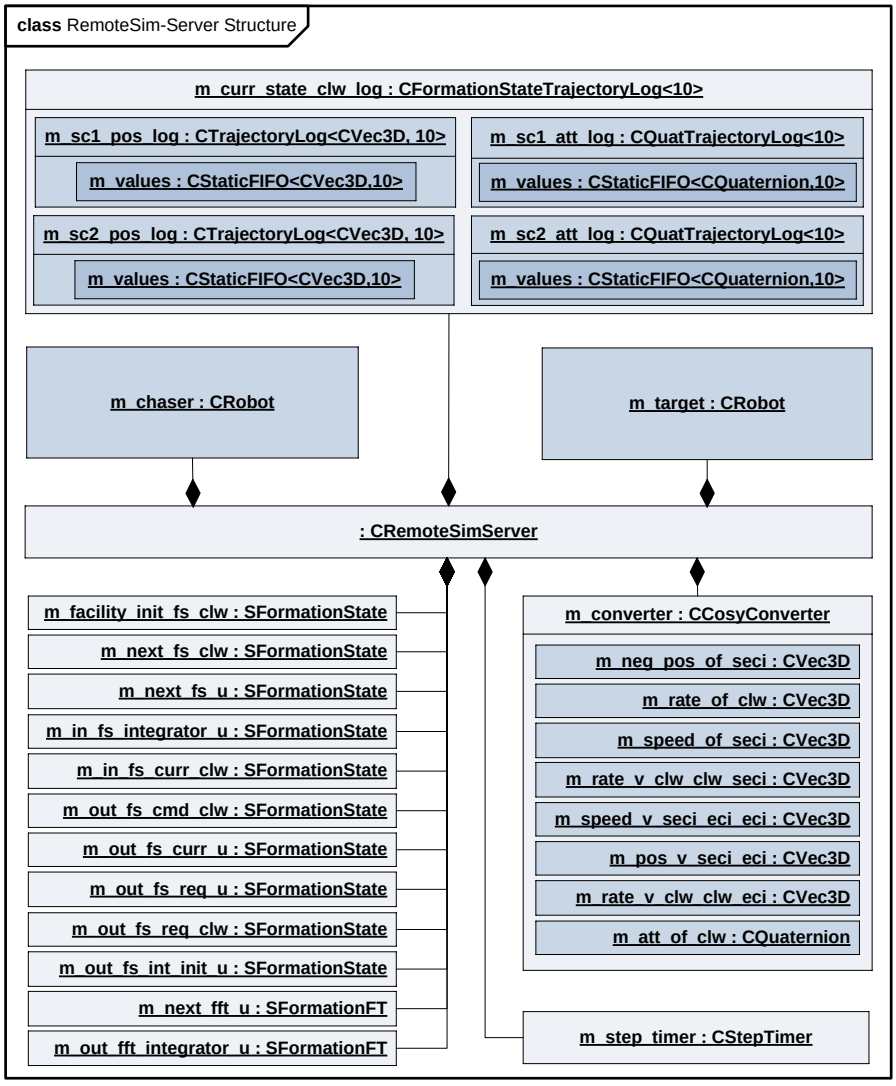

Figure 3.6.: UML Object Diagram of CRemoteSim Structure

ECI conversion thereafter.) `CFormationStateTrajectoryLog`, described in Sec.C.17, combines position and attitude logs (`CTrajectoryLog`, Sec.C.40 and `CQuatTrajectoryLog`, Sec.C.24) for both spacecraft. These values are used to calculate current speed and angular velocity and are given to the outputs expressed in U frame and in CLW frame. But the core components are represented by two instances of `CRobot`, one for the chaser `m_chaser`) and one for the target spacecraft (`m_target`). These "virtual" robots handle interpolation, keep station etc.. The class `CRobot` is presented in Fig.3.7.

There is a vector `m_direct_pos` and a quaternion `m_direct_att` which are used with `MODE_FORCE_TORQUE`. The position/attitude commands are taken from the integrator outputs directly and fed via class `CRobot` directly to the CMD outputs in that mode. With `MODE_TRAJECTORY`, interpolation is necessary. Translational and rotational movement has to be considered. The instance `m_pos_trajectory` of class `CCartTrajectoryGen` (like "cartesian", see Sec.C.3) deals with the former and the instance `m_att_trajectory` of class `CQuatTrajectoryGen` (like "quaternion", see Sec.C.23) with the latter. Each of the 3 translational degrees of freedom can be treated separately. Following this principle, `CCartTrajectoryGen` contains 3 instances of `CAtomicTrajectoryGen` (see Sec.C.2). Moving 1 level deeper in the class hierarchy, each instance of `CAtomicTrajectoryGen` contains a member variable `m_trajectory` (`CTrajectory`, see Sec.C.39) representing a 1D trajectory. In order to realize the interpolation algorithm outlined in Sec.2.6.4.5 including a linear equation system, the class `CLinEqusSys` (see Sec.C.14) is used. Moreover, the atomic trajectory generator

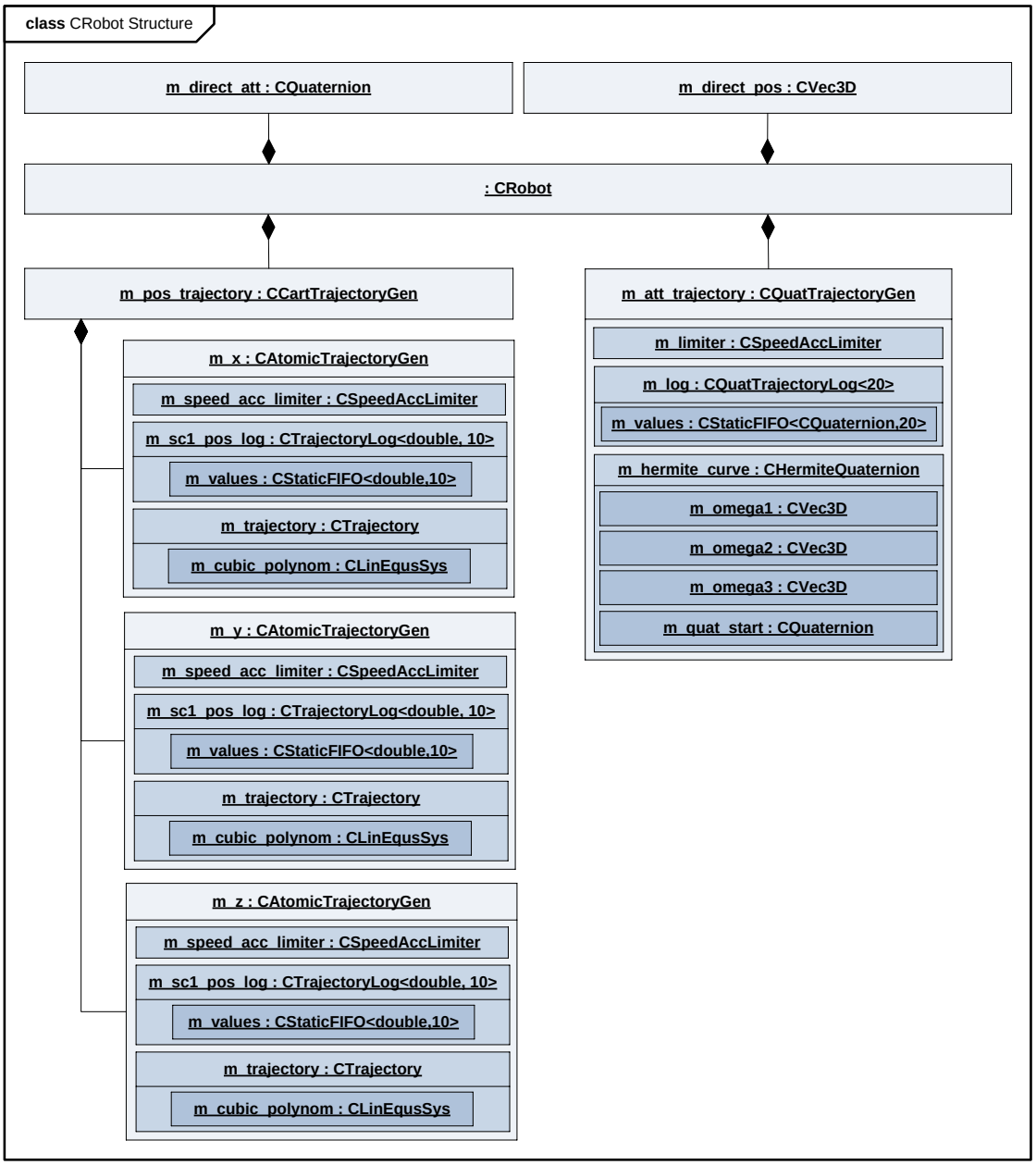

Figure 3.7.: UML Object Diagram of CRobot Structure

class comprises `m_speed_acc_limiter`. This instance of class `CSpeedAccLimiter` limits the robots' speed and acceleration and implements the keep station functionality (see Sec.C.33). And finally, one instance of `CTrajectoryLog` buffers the latest sequence of 1D trajectory values and calculates speed needed for interpolation. The structure of `CQuatTrajectoryGen` is quite similar to the one of `CAtomicTrajectoryGen`. There is a component for speed/acceleration limitation and a trajectory log (`CQuatTrajectoryLog`). But instead of `CTrajectory`, `CQuatTrajectoryGen` contains an instance of `CHermiteQuaternion` for C^1 continuous quaternion interpolation (see Sec.C.13). Thus, the quaternion algorithm presented in Sec.2.6.4.5 is realized.

3.4.3. Working Principle

As software structure already suggests, the RemoteSim-Server wrapper functions are also very similar to those of RemoteSim-Client. `remotesim_server_start_wrapper` instanciates and initializes `svr`, `remotesim_server_outputs_wrapper` is executed at each time step and `remotesim_server_-`

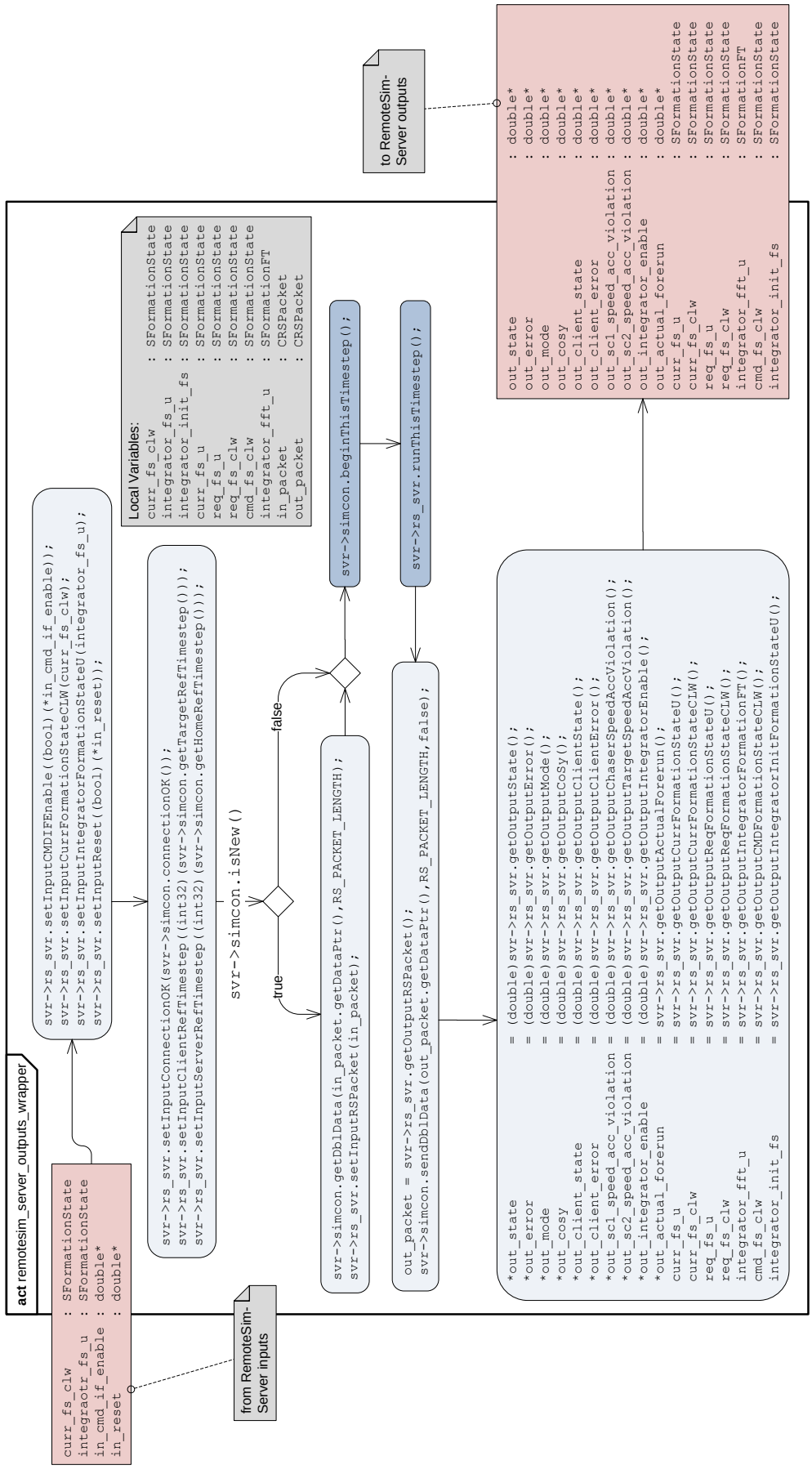

Figure 3.8.: UML Activity Diagram of the wrapper function `remotesim_server_outputs_wrapper`

outputs_wrapper destroys svr again. For the sake of completeness, the outputs wrapper routine is described in Fig.3.8, as the counterpart of Fig.3.3. Note the differences. svr has replaced cli and rs_svr has replaced rs_cli. Inputs and outputs are different. The overall structure, however, stays the same. The method CSimCon::beginThisTimestep() is presented in Sec.C.30. The method CRemoteSimServer::runThisTimestep() is depicted in the UML activity diagram Fig.C.30 (its client counterpart being CRemoteSimClient::runThisTimestep()). At first, the "virtual" robots (m_chaser and m_target) are informed about the actual positions and attitudes of the real robots. If the server is in STATE_FACILITY_INIT, then these positions and attitudes equal the values (given as RSP server parameters RSP_SVR_PARAM_CHASER_FACILITY_INIT_POS, RSP_SVR_PARAM_CHASER_FACILITY_INIT_ATT, RSP_SVR_PARAM_TARGET_FACILITY_INIT_POS and RSP_SVR_PARAM_TARGET_FACILITY_INIT_ATT) for facility initialization. In all other states, the commanded positions and attitudes of the last time step are used. This implies the assumption that the robots flawlessly behave as they are ordered. The alternative: Use current robot positions and attitudes as returned by the EPOS CMD interface. However, during developement, a considerable delay between commanding and associated position/attitude feedback occured. This principle delay would lead to instabilities in interpolation. Therefore, the author decided to use the commanded positions. At least, this leads to valid commanded trajectories for the robots. As soon as the robots' movements distinctly deviate from its commanded trajectories, interpolation might cause problems.

Next, the step timer (m_step_timer) is informed about this new time step by calling CStepTimer::beginThisTimestep. Also, the RSP server state machine, introduced in Sec.2.6.4.2, is executed (CRemoteSimServer::runStateMachine).

Finally, the positions and attitudes to be commanded to the EPOS CMD interface are set. There again is a conditional branch. If the CMD interface's enable signal is false, which means that the facility is not yet ready for synchronous commanding, the facility initial states (RemoteSim-Server parameters) are commanded. If the enable signal is true, positions and attitudes to be commanded are obtained from m_chaser and m_target by using the method CRobot::getNextPosAtt.

3.5. SimCon Implementation

The class CSimCon is an active class. Upon instanciation multiple threads are started, running in parallel, and implementing the various communication functions of the SCP protocol. Fig.3.9 shows CSimCon structure. There are four main components.

m_delay_manager, an instance of CDelayManager, reads SCP packet (CPacket) header fields and calculates the delay values defined in SCP (see Sec.2.5.7). In addition to these delays, derived values are computed for a better characterization. This is carried out by the corresponding instance of CDelay as member variable of m_delay_manager. For each of the delays Δt_{s2r}, Δt_{a2o}, Δt_{dead} and Δn_{dead}, mean, noise, jitter, standard deviation and Signal To Noise Ratio (SNR) are calculated. CDelayManager is described in Sec.C.12 and CDelay in Sec.C.11.

m_mem_manager, an instance of CSimConMemManager (see Sec.C.32), is a memory manager for dynamic memory allocation. The reason for invoking an additonal, custom memory manager is as follows. RemoteSim-Server needs to run on the EPOS ACS/RT which uses the real-time operating system VxWorks. During software developement, VxWorks showed to utilize a very primitive memory allocation algorithm. As a consequence, frequent allocation and deallocation leads to strong memory fragmentation in a short amount of time which inevitably results in memory allocation errors. (It took a considerable amount of time to figure out the cause for these errors.) However, it wouldn't have been practical to implement SCP without the possibility of allocating new SCP packets. Therefore the author chose to implement a basic memory manager which uses the VxWorks built-in memory management only once at program start-up. At this point, the working principle of the memory manager is outlined only briefly. At start-up, one contiguous block of memory is allocated. If some of the memory is needed, the block is split in two, a free block and a used block. The used block's size is as large as requested. If another portion of memory is needed, the free block is split accordingly and so forth. To avoid memory fragmentation, a defragmentation mechansim is implemented. Each time a used memory block is freed, CSimConMemManager checks whether the memory blocks located before and/or after the block to be freed are also free. In that case, the free blocks are combined. This way, free contiguous memory blocks are always as large as possible. This proved to be way ahead of VxWorks memory management capabilities.

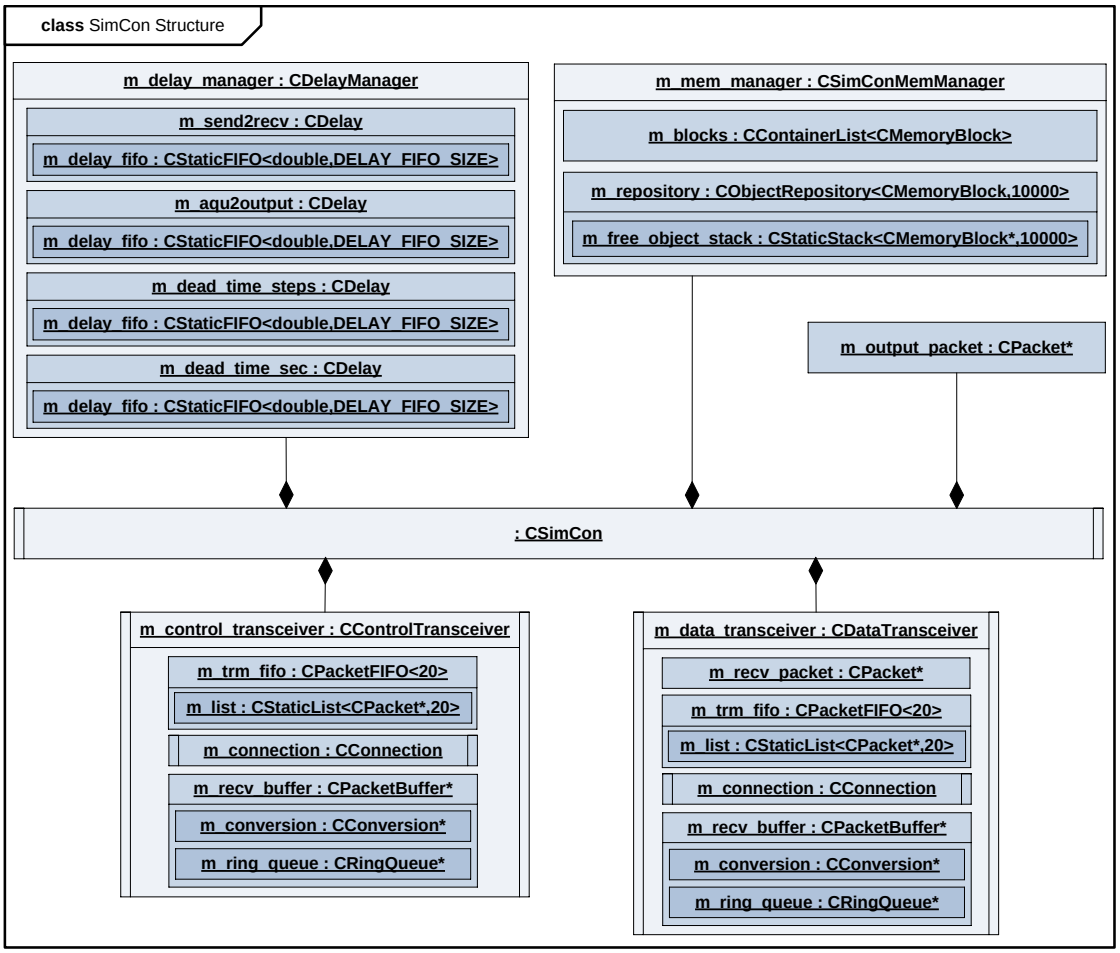

Figure 3.9.: UML Object Diagram of SimCon Software Structure

`m_control_transceiver`, an instance of `CControlTransceiver` (see Sec.C.9), realizes the SCP control connection. There is one thread transmitting ping packets in regular intervals and if the packets are not mirrored by the other SimCon instance in a certain amount of time, the connection is considered to be broken (`CControlTransceiver::pingThread()`). Another thread sends delay packets at regular intervals for calculation of the reference time (`CControlTransceiver::reftimeThread()`). This must not be confused with the reference packets sent at each time step to determine home and target reference time steps. This is also carried out by `CControlTransceiver`. Upon receipt of control connection packets, `m_control_transceiver` reacts as the Simulation Connection Protocol dictates (compare Sec.2.5).

`m_data_transceiver`, an instance of `CDataTransceiver` (see Sec.C.10), realizes the SCP data connection. It provides methods for transmission of data packets and it buffers the most recently received packet for processing. The `class` is thus simpler than `CControlTransceiver`.

Both, `CControlTransceiver` and `CDataTransceiver`, are inherited from `class CTransceiver` (see Sec.C.41), since functionalities like transmission of packets (`CTransceiver::sendThread()`) and receipt of packets (`CTransceiver::recvThread()`) are used by both classes.

In addition to these four componentents, there is also a `CPacket` (see Sec.C.19) which buffers the outgoing SCP packet.

4. Remote Control Analysis

4.1. A Demo Scenario

The simulation, the analysis in this chapter is based on, has not been carried out using the complete configuration depicted in Fig.1.6. Rather, a minimum configuration was chosen to illustrate proper simulation interconnection. It is not the purpose of this chapter to evaluate a specific FF simulation with associated algorithms etc., but to show that RemoteSim works as it is designed and that the principal setup - EPOS connected to a remote simulation via ethernet - is indeed capable of realizing a serious simulation. On the Formation-Flying side, the Simulink model running on the FCC constitutes the complete remote simulation. No other components are used. However, from perspective of the RemoteSim-Client, this makes no difference. The real-time behaviour of the simulation is ensured by synchronizing it with the WinXP clock. Since the sample time of the FF simulation is large ($1s$) the clock is sufficiently precise. Restrictions on the EPOS side of the simulation configuration are stronger. Instead of running the simulation on the real-time ACS/RT, it is executed in windows mode, also synchronized with the WinXP clock, using a dummy EPOS CMD interface block. The main reason for this is that the built-in logging blocks of Simulink allow to save only a very limited number of data sets in conjunction with Real-Time Workshop. A proper custom logging block was not available to the author at the time the data was acquired. Surely, this is a large difference to the actual remote simulation configuration. However, tests with real robot hardware have been conducted (without logging of data) successfully. From experience with these simulation tests, but without proof here, the author ensures that most of the logged values, trajectories etc. are also representative for a simulation run with the real-time ACS/RT. Results that are not representative are commented accordingly.

The simulation running on the FCC (mainly) comprises the RemoteSim-Client and an enabled subsystem containing the FF model. In this simulation, for simplicity, the chaser's position and speed is set to 0 and the FF controller steers the target spacecraft such that the initial relative distance is zero and is subsequently enlarged slowly. This is no realistic scenario of course, but ideal for testing due to significant key points (same position at the beginning of simulation). Furthermore, it should be noted that the only thing that is important is the *relative* position of both spacecraft. Moreover, attitude is constant for both spacecraft.

Tab.4.1 shows RemoteSim-Client parameters for this simulation.

Table 4.1.: Parameter Settings of RemoteSim-Client Simulink Block

Parameter	Setting
Control Port	3000
Data Port	3001
IP of RemoteSim-Server	172.16.247.7
Connection Timeout	4000
Mode of Operation	Command Trajectory
Coordinate Mapping	Feedthrough

Continued on next page

Table 4.1 – *Continued from previous page*

Parameter	Setting
Client Forerun	0.5
Timespan for Reaching Initial Conditions	20
Sampletime	1

The trajectory is commanded. There is no coordinate transformation to be carried out by RemoteSim-Server. The FF simulation commands in relative coordinates already.

The EPOS part of the simulation configuration is comprised of a RemoteSim-Server block and a replacement for the EPOS CMD interface which essentially introduces some delay between commanded and returned (current) robot positions. The CMD interface enable signal is controlled with a manual switch. In general, the simulation running on the EPOS side will always be of similar principle structure. Tab.4.2 summarizes RemoteSim-Server settings.

Table 4.2.: Parameter Settings of RemoteSim-Server Simulink Block

Parameter	Setting
Control Port	3000
Data Port	3001
IP of RemoteSim-Client	172.16.247.100
Connection Timeout	4000
Speed Limit	1000
Acceleration Limit	0.1
Angular Speed Limit	1000
Angular Acceleration Limit	0.1
Translational Braking Acceleration	0.1
Rotational Braking Acceleration	0.1
Initial Position Chaser	[0.4 0 0]
Initial Position Target	[8 0 0]
Initial Attitude Chaser	[0 0 0 1]
Initial Attitude Target	[0 0 0 1]
Sampletime	0.004

Note that data and control port match those specified for RemoteSim-Client. Speed and angular speed limit are set to a high value which switches off these limits. Practically, these settings have

no meaning for this simulation. There are no large velocities to be expected. Acceleration and angular acceleration are limited to 0.1 for safety. (Even at low speeds, a sudden acceleration in an unexpected fashion may damage hardware.) Translational and rotational breaking acceleration are set to reasonable values from experience. The sample time of 0.004 is demanded by the EPOS facility.

4.2. State Transitions

Fig.4.1 shows server and client state during the simulation. In fact, server state is read from

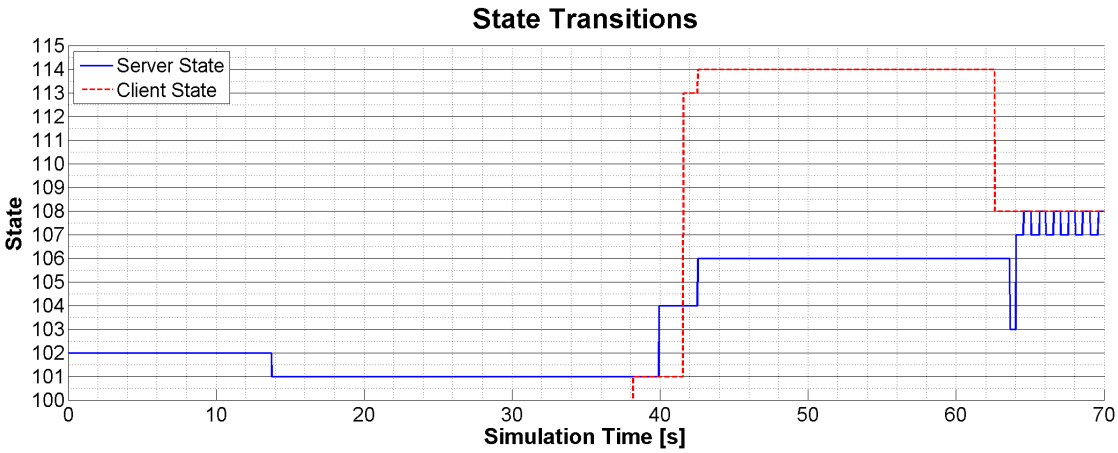

Figure 4.1.: Plot of Server and Client State Transitions as read from RemoteSim-Server Outputs

RemoteSim-Server output "State" and client state from RemoteSim-Server output "Client State". Note that data logging has not been carried out with a sample frequency of $250Hz$ but with $50Hz$, to keep the amount of data reasonably small. As a consequence, a peak may not show on the plot. The association between number and actual state is the same as shown in Fig.2.16. (That's how the states are implemented in the code.) The purpose of Fig.4.1 is to confirm that RSP server/client interaction as designed in theory in Sec.2 also works practically. Instead of recapitulating the whole interaction again, only a few key points shall be mentioned here. First, the order of states is similar to Fig.2.15 and Fig.2.16. For the server: Wait for finish of facility initilization (102), safe slow down (105, not visible, since only one time step long), wait for a valid connection (101), wait for the init CMD (104), await the first trajectory CMD during initialization (106), wait for reaching initial time step (103) and then by turns await the next CMD (107) and wait for next hit (108). For the client: wait for a valid connection (101), wait until the server is ready for an init CMD (113), initial trajectory (114) and finally for the rest of the simulation trajectory commanding (108). The key interaction point is when the server becomes ready for receipt of an init CMD (change from 101 to 104) and the client reacts by transmission of this CMD and transition to initial trajectory mode (change from 113 to 114). This is all consistent with the design presented in Sec.2.

4.3. Facility Initialization and Initial Trajectory

In Fig.4.2 target spacecraft position during the initial trajectory is depicted. To pretain a good overview, only the x component is shown, representative for the trajectory. The plot comprises three curves. The requested position, provided by RemoteSim-Server output Target Requested State (CLW), the commanded position, provided by output Target Command Position (CLW), and the current position, provided by output Target Current State (CLW). The simulation is configured (RemoteSim-Client settings) such that no coordinate transformation is carried out by RemoteSim. Hence, values in U and CLW are equal. Therefore, only CLW coordinates are considered here. The plot starts at $t = 0$, e.g. when the server simulation is started. At the beginning, commanded position equals the one for facility initialization given as RemoteSim-Server parameter ($[8; 0; 0]$). Current position follows the commanded position immediately. There is no movement of the robots during facility initialization as

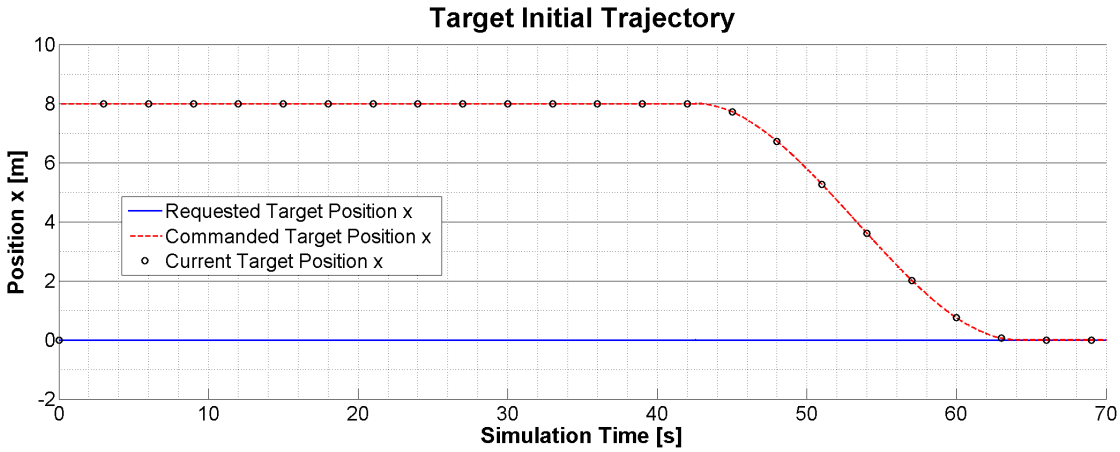

Figure 4.2.: Plot of Target Position (x-component) during Initial Trajectory

would be expected with the Move to Start phase. This is due to the fact that a dummy EPOS CMD interface is used to allow the server simulation run on WinXP. As mentioned in the previous section, requested position is 0 at the beginning of the simulation. As soon as RemoteSim-Server receives the init CMD, the target starts to follow the initial trajectory which approaches the requested position (and speed) at facility initialization smoothly and asymptotically. It is important to note that there are no edges in the trajectory.

4.4. Continuity of Interpolation

In Fig.4.3 target position is depicted for a small time interval during regular simulation, after the initial trajectory. Requested speed changes every second, consistent with RemoteSim-Client sample

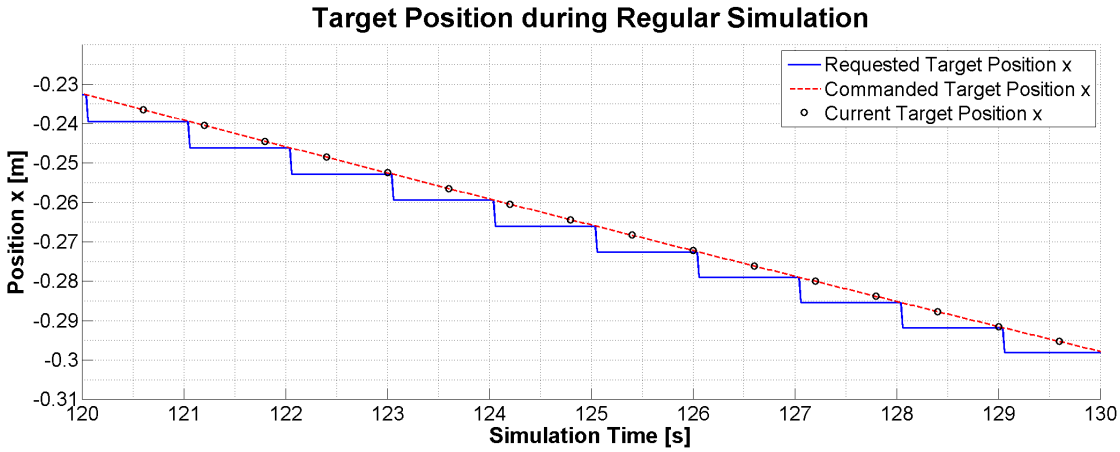

Figure 4.3.: Plot of Target Position (x-component) during regular Simulation

frequency of $1Hz$. The interpolated trajectory which is commanded follows the requested one precisely and smoothly. There are no peaks or undesirable curving between simulation points. That the interpolated trajectory is indeed C^1 continuous becomes clear when looking at the associated speed plot, depicted in Fig.4.4. Requested and current speed are compared. (There is no commanded speed. Current speed is calculated from the sequence of returned actual positions which ideally equal the commanded positions.) Also speed follows the requested values. There are no jumps in velocity and therefore the first derivative of position is continuous as required.

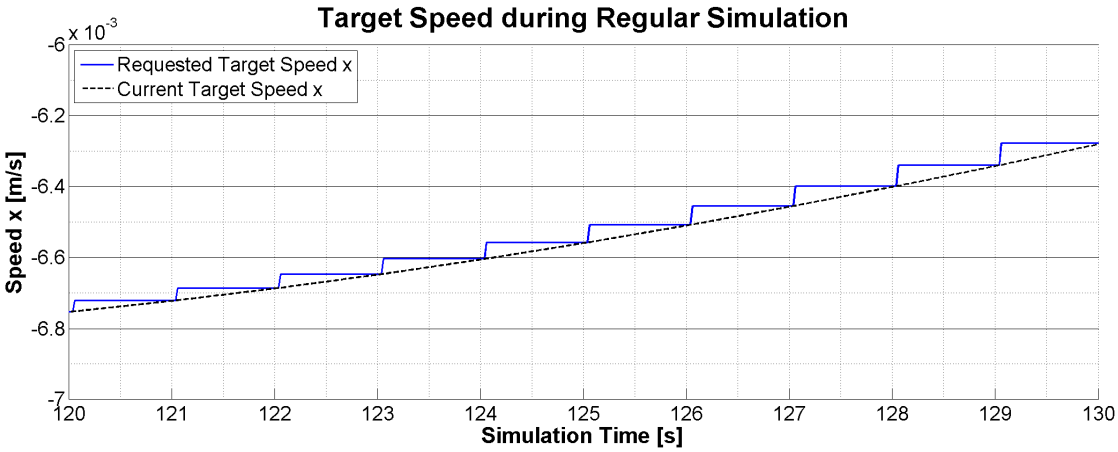

Figure 4.4.: Plot of Target Speed (x-component) during regular Simuation

4.5. Connection Quality

An important feature of SCP is the capability to monitor connection quality in the form of several different time delays concerning packet transmission and receipt. Sec.2.5 gives a detailed description of these delays. In this section, the demo scenario is used to present data which allows to assertain the suitability of the local EPOS network for realizing a remote simulation via ethernet. The data is obtained from the according RemoteSim-Server outputs. The data presented in this section was obtained by running a WinXP simulation of RemoteSim-Server. However, it is representative for a simulation with the RTOS VxWorks also. The network (switches, cables) is the same and, due to using equal PCs, the network controllers are the same also. Moreover, experience with simulations carried out with VxWorks (but not logged) showed that the data is comparable.

Fig.4.5 shows the Send to Receive delay (Δt_{s2r}) and its mean (calculated at each point with the last 20 values). This delay represents the time between transmission of the packet by one SimCon instance

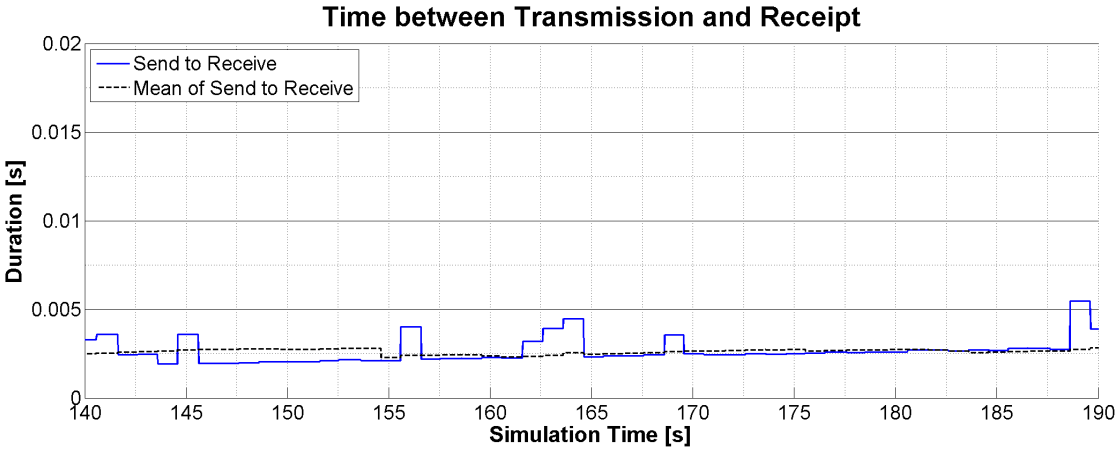

Figure 4.5.: Plot of Time between Transmission and Receipt of a Packet

(here the remote simulation, e.g. RemoteSim-Client) and receipt by the other SimCon instance (here the EPOS simulation, e.g. RemoteSim-Server). As expected, there are statistical variations. The mean value is about $2.5ms$. This is even below EPOS sample time. Note also that changes occur at a $1s$ rythm, consistent with RemoteSim-Client sample frequency of $1Hz$.

Compared to Send to Receive, the delay Acquisition to Output (Δt_{a2o})depicted in Fig.4.6 is considerably larger. This delay represents the time between acquisition of data from the inputs of RemoteSim-Client and the provision of the data to the outputs of RemoteSim-Server. Not only is the mean larger, namely $20ms$, but also the variation is more distinct. This can be ascribed to two main reasons. Acquistion to Output also includes the processing time of the packet by the software. Various threads

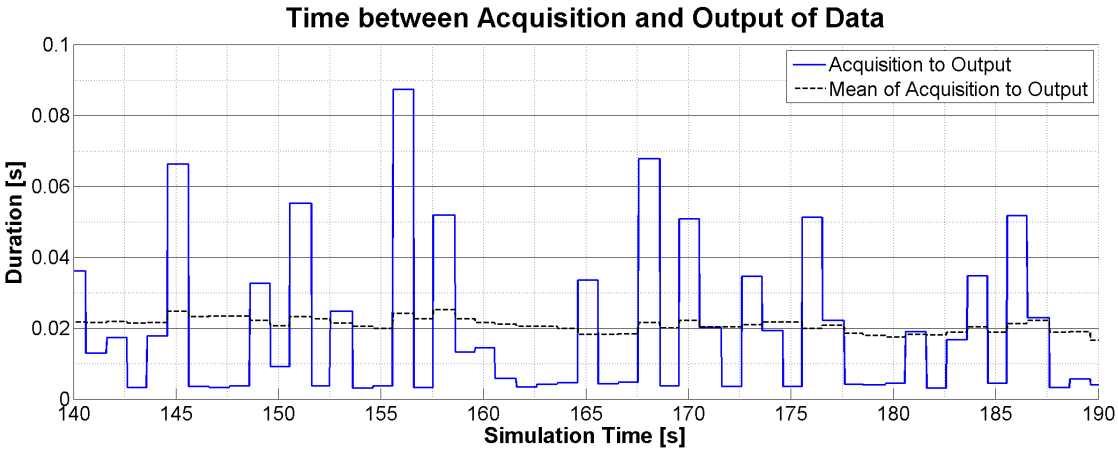

Figure 4.6.: Plot of Time between Acquisition and Output of Packet Data

are at work in parallel. Therefore, processing time may vary to some degree. Moreover, acquisition of data and output of data can only be carried at discrete time steps. At some point, server and client time steps may be located temporally closer and at some point wider. This is closely related to variation of the actual forerun (see Sec.2.6.2). Jitter and standard deviation confirm this observation. Both values are depicted in Fig.4.7 for Send to Receive and in Fig.4.8 for Acquisition to Output. In the former case, standard deviation lies between $0.001s$ and $0.0025s$, while in the latter case, it is about $0.02s$.

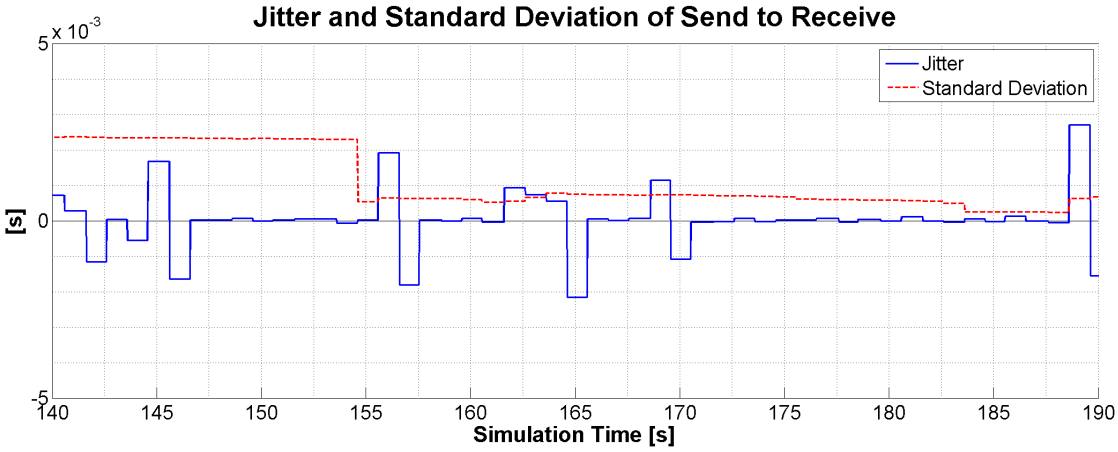

Figure 4.7.: Plot of Jitter and Standard Deviation of Send to Receive

Another important delay value is deadtime (Δt_{dead}), depicted in Fig.4.9 together with its mean. It represents the time between acquisition of data by RemoteSim-Server (forward packet) and output of the data a specific received packet contains (return packet). It is the packet RemoteSim-Client has sent first after having received the aforementioned packet from RemoteSim-Server. Thus, deadtime is a measure for the delay between a general command of some sort and receipt of some kind of feedback information related to this command. It is the deadtime, in the sense of control theory, the connection introduces in the simulation system. The mean deadtime shown in Fig.4.9 is about $0.04s$ which equals about twice the mean of Acquisition to Output. This makes sense from a logical point of view, since two times the process of acquisition and output of data is involved. The standard deviation of deadtime depicted in Fig.4.10 equals about the standard deviation of Acquistion to Output.

Deadtime in steps is not presented here. In principle, it equals deadtime in seconds but expressed in full simulation steps. Beside that, the author encountered a minor bug in calculation of deadtime in steps which he was not able to fix up to the time the data was acquired for this chapter. However, all characteristic delay values are covered with the above presented data.

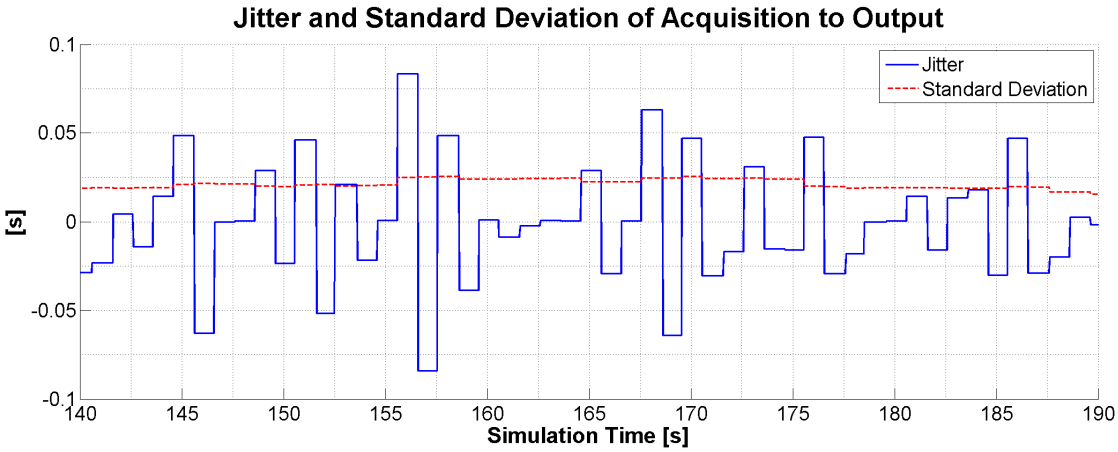

Figure 4.8.: Plot of Jitter and Standard Deviation of Acquisition to Output

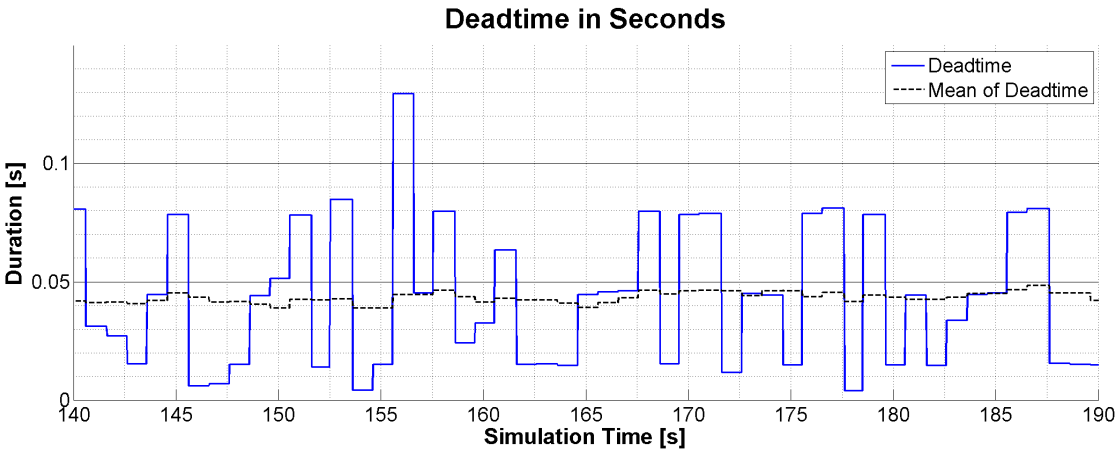

Figure 4.9.: Plot of Deadtime

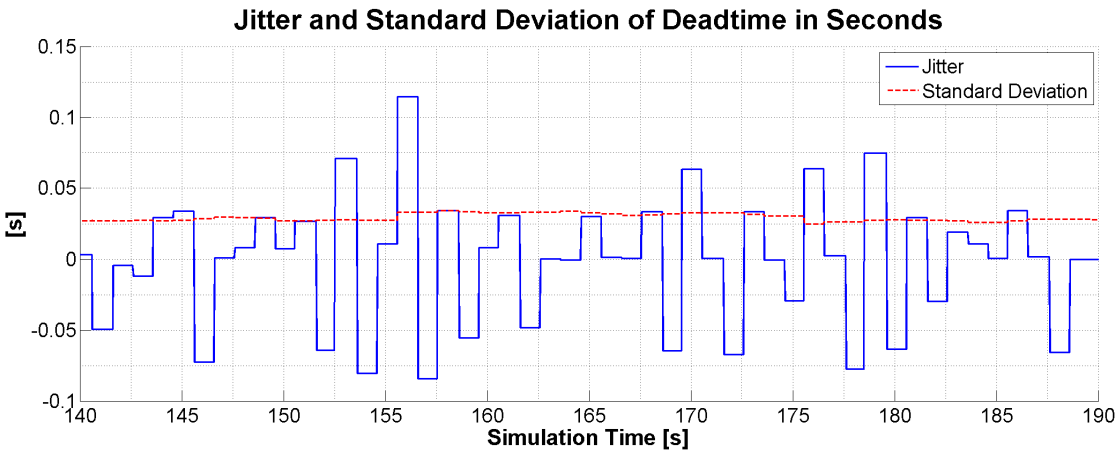

Figure 4.10.: Plot of Jitter and Standard Deviation of Acquisition to Output

In order to appraise the impact of the presented data on the simulation process, consider the range of reasonable RemoteSim-Client sample frequencies. The FF-Testbed presented in Sec.1.3.2 works with a frequency of at most $10Hz$. Other remote simulations running flight software will work in that area of sample frequency, too. Hence, the delay times in this section, especially a mean Acquisiton to Output delay of $20ms$, are distinctly below critical values for the simulation system. Individual peaks can be dealt with by extrapolation (see Sec.2.6.5).

4.6. Some Comments on Drift

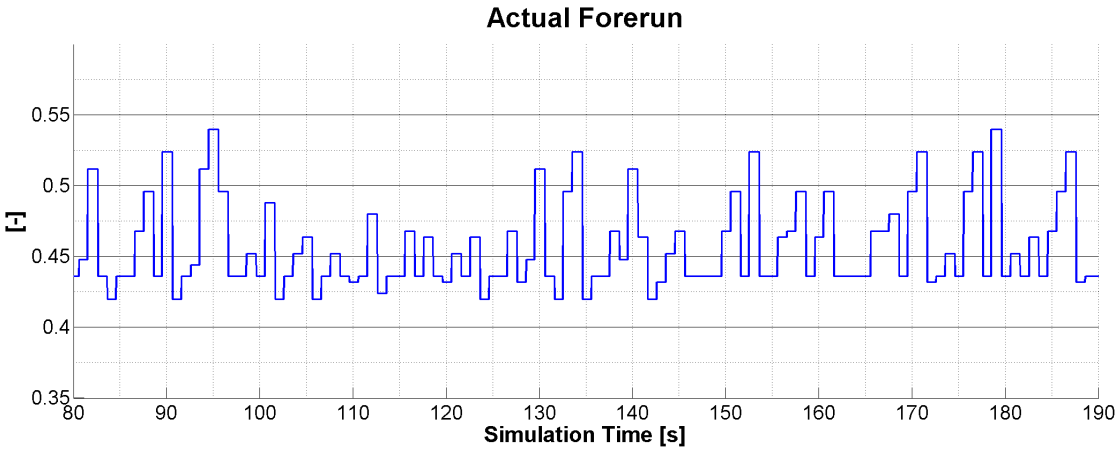

Figure 4.11.: Actual Forerun

Drift is the time differential between the remote simulation and the EPOS simulation. It is different from zero if the clocks the simulations are synchronized with don't run precisely with equal speed. The tolerance of electronic components alone dictates that a drift is to be expected. With RemoteSim, a measure for drift is the change in actual forerun, introduced in Sec.2.6.2. Fig.4.11 depicts the actual forerun during a portion of the simulation. This Fig. is not representative for a real-time simulation (using the ACS/RT). However, from manual observation, but without proof here, the actual forerun showed to be about $10^{-6}s/s$. This means that after $1000000s$ a drift of $1s$ can be measured. This is well within the order of magnitude of a quartz crystal's precision (see [18]). Since software depends on the clock, drift cannot be reduced further.

5. Conclusion

In this thesis, a software package has been developed that allows to connect an external satellite simulator via ethernet with the EPOS real-time control system, thereby reducing the adaption effort for external customers and introducing new functionalities.

Requirement analysis showed two main elements that have to be dealt with on the top-level:

- Provide a bidirectional data connection between two Simulink models on different hosts, including monitoring of connection quality and status.
- Manage the interaction between remote simulation and EPOS, including realization of starting conditions/initial trajectory, seamless transition to simulation and general timing.

This distribution of tasks and responsibilities led to the design of two communication protocols, which form a network application layer protocol. The Simulation Connection Protocol (SCP) provides the data connection, the Remote Simulation Protocol (RSP) manages simulation interaction, where RSP relies on the services of SCP, complient with the layer concept of computer networks.

The need to monitor connection quality showed to require some kind of mechanism for determining the simulation time of the connected Simulink model. In order to measure the time it took a received data packet from origin to destination, some kind of information about the correlation between the origin's simulation time and the target's simulation time is needed. Due to the statistical nature of packet-based communications, design of this mechanism was not trivial. It relies upon regular transmission and mirroring of specific packets and averaging of the resulting transmission times. This mechanism dictates the structure of a SCP packet, including various header fields used for delay calculation.

The given boundary conditions for simulation interaction demanded that there be no "hard" synchronization between remote simulation and EPOS, "hard" meaning a separate synchronization signal. As a result, RSP was designed to realize a strictly timed simulation interaction without relying on such a signal. This includes exchange of reference time steps (provided by SCP) to manage timing, on-the-fly determination of initial time step and interpolation intervals as well as concepts like forerun (to have a margin for statistically delayed network packets) and interpolation hits.

Both communication protocols were implemented as Simulink c-mex S-Function blocks. The remote simulation has to contain a RemoteSim-Client block and the EPOS simulation a RemoteSim-Server block. These RemoteSim blocks comprise both protocols. Implementation for Windows proofed to be widely unproblematic. Only two noteworthy issues shall be mentioned here. The resolution of the standard time function `clock()` is not large enough ($16ms$) for serious determination of packet delays. Rather the (platform-dependent) so-called `QueryPerformanceTimer` has to be used. The other issue addresses the use of an enabled subsystem for the user simulation model as part of the external simulator. RemoteSim-Client uses an enable signal to start the actual remote simulation. This means, that the complete model of the customer has to be placed in such an enabled subsystem. Experiments with the Formation-Flying Simulink library showed that many custom Simulink blocks rely on the simulation time as an decisive value. However, in general, the enabled subsystem is activated some time after simulation start. As a consequence, the actual simulation time is different from the time passed for the enabled subsystem. This must be accounted for when designing a Simulink model for a remote simulation. In contrast, implementation for VxWorks came about to be problematic at some points. First, debugging was rather difficult with VxWorks, since bugs in the code resulting in memory access violations immediately led to a crash of the whole system, making debugging impossible. Moreover, in a real-time environment like VxWorks output functions like `printf()` have to be used with care. They take a considerable amount of time for execution. And at a sample time of not more than $4ms$ for each time step, there is not much room for such time consuming functions. The biggest problem, however, had its origin in VxWorks' dynamic memory management. It uses only a very simple memory allocation algorithm leading to memory fragmentation rapidly. A reasonable implementation of SCP requires dynamic memory allocation. Therefore, the author had to design

a custom memory manager including a simple defragmentation algorithm. Moreover, many parts of the software had to be programmed such that dynamic memory allocation is avoided where possible. Also, this problem prevented the author from using many parts of the C++ standard template library.

A Formation-Flying demo scenario was simulated with RemoteSim. Evaluation of state transitions, realization of initial trajectory and interpolation showed that the remote simulation software developed in this theses fullfills the requirements. Interpolation even tested with a sample frequency of $1Hz$, e.g. interpolation intervals of $1s$, turned out to be absolutely smooth and continuous. External simulators running at even such low frequencies can work together with EPOS. Analysis of connection quality showed a mean delay of $20ms$ including processing from data acquisition at the remote simulation to data output at EPOS. Considering a maximum sample frequency of $10Hz$, this delay is far below critical values. Drift between the FF-Testbed and the EPOS ACS/RT was within the limits of standard hardware timer precision ($10^{-6}\,s/s$). As a result, several hours of continuous simulation are possible before drift becomes a problem. RemoteSim, as a software package, cannot reduce drift any further, with the given boundary conditions.

In the future, it might be benificial to extend RemoteSim with a synchronization mechanism via ethernet. A special packet could be used to serve as a synchronization signal. EPOS would provide the reference time. This would obliterate the drift problem altogether and allow even longer simulations. Another interesting idea is to not only use the EPOS local network but to place the remote simulation in another building or even farther away. Thorough examinations would be necessary concerning packet delay and suitability of the connection for real-time simulation (with an appropriately low sample frequency).

In summary, additional functionalities have been added to EPOS with the remote simulation software developed in this theses. Up till now, a simulation running directly on the EPOS real-time computer, was restricted to an initial speed and initial angular velocity of zero ("Move to Start") for the robots/spacecraft. Now, using a remote simulation connected via ethernet, both initial speed and initial angular velocity can be realized in the form of an automatically calculated initial trajectory. Moreover, it is possible to restart the remote simulation or even a different remote simulation multiple times without stopping the EPOS real-time simulation (which would include carrying out several steps of the synchronous CMD procedure). The robots are braked automatically, so that another initial trajectory can be calculated as a starting point for another simulation. Also, the tough restriction of a sample frequency of $250Hz$ is circumvented by interpolation and extrapolation, if required.

In this way, the remote simulation system RemoteSim, designed, implemented and tested in this thesis contributes to the flexibility and operational capabilities of the Hardware-in-the-Loop Rendezvous and Docking simulator EPOS.

A. RemoteSim-Client Block Parameters, Inputs and Outputs

Table A.1.: Inputs of RemoteSim-Client Simulink Block. A description is given in Tab.2.5 via the associated RSP client inputs. The data type is double for all input ports. See Sec.1.5 for quaternion definition.

Input	Associated RSP client Input	Unit	Format
Chaser Requested Position (U)	RSP_CLI_IN_CHASER_REQ_POS	m	$(x\,y\,z)$
Chaser Requested Speed (U)	RSP_CLI_IN_CHASER_REQ_SPEED	m/s	$(v_x\,v_y\,v_z)$
Chaser Requested Attitude (U)	RSP_CLI_IN_CHASER_REQ_ATT	-	$(q_1\,q_2\,q_3\,q_0)$, q_0 scalar part
Chaser Requested Rate (U)	RSP_CLI_IN_CHASER_REQ_RATE	rad/s	$(\omega_x\,\omega_y\,\omega_z)$
Target Requested Position (U)	RSP_CLI_IN_TARGET_REQ_POS	m	$(x\,y\,z)$
Target Requested Speed (U)	RSP_CLI_IN_TARGET_REQ_SPEED	m/s	$(v_x\,v_y\,v_z)$
Target Requested Attitude (U)	RSP_CLI_IN_TARGET_REQ_ATT	-	$(q_1\,q_2\,q_3\,q_0)$, q_0 scalar part
Target Requested Rate (U)	RSP_CLI_IN_TARGET_REQ_RATE	rad/s	$(\omega_x\,\omega_y\,\omega_z)$
Chaser Requested Force	RSP_CLI_IN_CHASER_REQ_FORCE	N	$(F_x\,F_y\,F_z)$
Chaser Requested Torque	RSP_CLI_IN_CHASER_REQ_TORQUE	Nm	$(T_x\,T_y\,T_z)$
Target Requested Force	RSP_CLI_IN_TARGET_REQ_FORCE	N	$(F_x\,F_y\,F_z)$
Target Requested Torque	RSP_CLI_IN_TARGET_REQ_TORQUE	Nm	$(T_x\,T_y\,T_z)$

Table A.2.: Outputs of RemoteSim-Client Simulink Block. A description is given in Tab.2.6 via the associated RSP client outputs and in Tab.2.3 via the associated SCP outputs. The data type is double for all output ports. See Sec.1.5 for quaternion definition.

Output	Associated RSP client/SCP Output	Unit	Format
Chaser Current Position (U)	RSP_CLI_OUT_CHASER_CURR_POS	m	$(x\,y\,z)$
Chaser Current Speed (U)	RSP_CLI_OUT_CHASER_CURR_SPEED	m/s	$(v_x\,v_y\,v_z)$
Chaser Current Attitude (U)	RSP_CLI_OUT_CHASER_CURR_ATT	-	$(q_1\,q_2\,q_3\,q_0)$, q_0 scalar part
Chaser Current Rate (U)	RSP_CLI_OUT_CHASER_CURR_RATE	rad/s	$(\omega_x\,\omega_y\,\omega_z)$
Target Current Position (U)	RSP_CLI_OUT_TARGET_CURR_POS	m	$(x\,y\,z)$

Continued on next page

Table A.2 – *Continued from previous page*

Output	Associated RSP client/SCP Output	Unit	Format
Target Current Speed (U)	RSP_CLI_OUT_TARGET_CURR_SPEED	m/s	$(v_x\ v_y\ v_z)$
Target Current Attitude (U)	RSP_CLI_OUT_TARGET_CURR_ATT	-	$(q_1\ q_2\ q_3\ q_0)$, q_0 scalar part
Target Current Rate (U)	RSP_CLI_OUT_TARGET_CURR_RATE	rad/s	$(\omega_x\ \omega_y\ \omega_z)$
Server State	RSP_CLI_OUT_SERVER_STATE	-	scalar
Server Error	RSP_CLI_OUT_SERVER_ERROR	-	scalar
State	RSP_CLI_OUT_STATE	-	-
Error	Currently, RSP does not povide for any error on the client side. This output is for future extensions.		
Simulation Enable	RSP_CLI_OUT_SIM_ENABLE	-	scalar
Send 2 Receive	SCP_OUT_SEND2RECV	s	$(\Delta t_{s2r}\ \overline{\Delta t_{s2r}}\ \Delta t_{s2r,noise}\ \Delta t_{s2r,jitter}\ \Delta t_{s2r,stddev}\ \Delta t_{s2r,SNR})$
Acquisition 2 Output	SCP_OUT_ACQU2OUTPUT	s	$(\Delta t_{a2o}\ \overline{\Delta t_{a2o}}\ \Delta t_{a2o,noise}\ \Delta t_{a2o,jitter}\ \Delta t_{a2o,stddev}\ \Delta t_{a2o,SNR})$
Deadtime in sec	SCP_OUT_DEADTIME_SEC	s	$(\Delta t_{dead}\ \overline{\Delta t_{dead}}\ \Delta t_{dead,noise}\ \Delta t_{dead,jitter}\ \Delta t_{dead,stddev}\ \Delta t_{dead,SNR})$
Deadtime in steps	SCP_OUT_DEADTIME	-	$(\Delta n_{dead}\ \overline{\Delta n_{dead}}\ \Delta n_{dead,noise}\ \Delta n_{dead,jitter}\ \Delta n_{dead,stddev}\ \Delta n_{dead,SNR})$
Connection OK	SCP_OUT_CONNECTION_OK	-	scalar
Client Reference Timestep	SCP_OUT_HOME_REF_TIMESTEP	-	scalar
Server Reference Timestep	SCP_OUT_TARGET_REF_TIMESTEP	-	scalar

Table A.3.: Parameters of RemoteSim-Client Simulink Block. A description is given in Tab.2.4 via the associated RSP client parameters and in Tab.2.1 via the associated SCP parameters.

Parameter	Associated RSP client/SCP Parameter	Unit	Format	Example
Port for Control Connection	SCP_PARAM_CONTROL_PORT	-	scalar integer $0-65535$	3000
Port for Data Connection	SCP_PARAM_DATA_PORT	-	scalar integer $0-65535$	3001
IP of Target Machine	SCP_PARAM_IP	-	string	'172.16.247.5'
Connection Timeout	SCP_PARAM_TIMEOUT	ms	scalar integer	4000
Mode of Commanding	RSP_CLI_PARAM_MODE	-	drop down list	Trajectory

Continued on next page

Table A.3 – *Continued from previous page*

Parameter	Associated RSP client/SCP Parameter	Unit	Format	Example
Coordinate Transformation	`RSP_CLI_PARAM_COSY`	-	drop down list	Feedthrough
Desired Forerun	`RSP_CLI_PARAM_FORERUN`	-	scalar double $0.0 - 1.0$	0.5
Timespan for Initial Trajectory	`RSP_CLI_PARAM_INIT_TIMESPAN`	s	scalar double	60
Sampletime	`RSP_CLI_PARAM_SAMPLETIME`	s	scalar double	1

B. RemoteSim-Server Block Parameters, Inputs and Outputs

Table B.1.: Inputs of RemoteSim-Server Simulink Block. A description is given in Tab.2.10 via the associated RSP server inputs. The data type is double for all input ports. See Sec.1.5 for quaternion definition.

Input	Associated RSP server Input	Unit	Format
CMD Inteface Enable	RSP_SVR_IN_CMD_IF_ENABLE	-	scalar
Chaser Current Position (CLW)	RSP_SVR_IN_CHASER_CURR_POS	m	$(x\,y\,z)$
Chaser Current Speed (CLW)	RSP_SVR_IN_CHASER_CURR_ATT	-	$(q_1\,q_2\,q_3\,q_0)$, q_0 scalar part
Target Current Position (CLW)	RSP_SVR_IN_TARGET_CURR_POS	m	$(x\,y\,z)$
Target Current Attitude (CLW)	RSP_SVR_IN_TARGET_CURR_ATT	-	$(q_1\,q_2\,q_3\,q_0)$, q_0 scalar part
Chaser Integrator Position (U)	RSP_SVR_IN_CHASER_INT_POS	m	$(x\,y\,z)$
Chaser Integrator Speed (U)	RSP_SVR_IN_CHASER_INT_SPEED	m/s	$(v_x\,v_y\,v_z)$
Chaser Integrator Attitude (U)	RSP_SVR_IN_CHASER_INT_ATT	-	$(q_1\,q_2\,q_3\,q_0)$, q_0 scalar part
Chaser Integrator Rate (U)	RSP_SVR_IN_CHASER_INT_RATE	rad/s	$(\omega_x\,\omega_y\,\omega_z)$
Target Integrator Position (U)	RSP_SVR_IN_TARGET_INT_POS	m	$(x\,y\,z)$
Target Integrator Speed (U)	RSP_SVR_IN_TARGET_INT_SPEED	m/s	$(v_x\,v_y\,v_z)$
Target Integrator Attitude (U)	RSP_SVR_IN_TARGET_INT_ATT	-	$(q_1\,q_2\,q_3\,q_0)$, q_0 scalar part
Target Integrator Rate (U)	RSP_SVR_IN_TARGET_INT_RATE	rad/s	$(\omega_x\,\omega_y\,\omega_z)$
Reset	RSP_SVR_IN_RESET	-	scalar

Table B.2.: Outputs of RemoteSim-Server Simulink Block. A description is given in Tab.2.11 via the associated RSP server outputs and in Tab.2.3 via the associated SCP outputs. The data type is double for all output ports. See Sec.1.5 for quaternion definition.

Output	Associated RSP server/SCP Output	Unit	Format
State	RSP_SVR_OUT_STATE	-	scalar
Error	RSP_SVR_OUT_ERROR	-	scalar
Mode	RSP_SVR_OUT_MODE	-	scalar

Continued on next page

Table B.2 – *Continued from previous page*

Output	Associated RSP server/SCP Output	Unit	Format
Cosy	RSP_SVR_OUT_COSY	-	scalar
Client State	RSP_SVR_OUT_CLIENT_STATE	-	scalar
Client Error	Currently, RSP does not povide for any error on the client side. This output is for future extensions.		
Chaser Current State (U)	RSP_SVR_OUT_CHASER_CURR_POS_U RSP_SVR_OUT_CHASER_CURR_SPEED_U RSP_SVR_OUT_CHASER_CURR_ATT_U RSP_SVR_OUT_CHASER_CURR_RATE_U	-	$(x\ y\ z\ v_x\ v_y\ v_z$ $q_1\ q_2\ q_3\ q_0\ \omega_x\ \omega_y\ \omega_z)$ q_0 scalar part
Target Current State (U)	RSP_SVR_OUT_TARGET_CURR_POS_U RSP_SVR_OUT_TARGET_CURR_SPEED_U RSP_SVR_OUT_TARGET_CURR_ATT_U RSP_SVR_OUT_TARGET_CURR_RATE_U	-	$(x\ y\ z\ v_x\ v_y\ v_z$ $q_1\ q_2\ q_3\ q_0\ \omega_x\ \omega_y\ \omega_z)$ q_0 scalar part
Chaser Current State (CLW)	RSP_SVR_OUT_CHASER_CURR_POS_CLW RSP_SVR_OUT_CHASER_CURR_SPEED_CLW RSP_SVR_OUT_CHASER_CURR_ATT_CLW RSP_SVR_OUT_CHASER_CURR_RATE_CLW	-	$(x\ y\ z\ v_x\ v_y\ v_z$ $q_1\ q_2\ q_3\ q_0\ \omega_x\ \omega_y\ \omega_z)$ q_0 scalar part
Target Current State (CLW)	RSP_SVR_OUT_TARGET_CURR_POS_CLW RSP_SVR_OUT_TARGET_CURR_SPEED_CLW RSP_SVR_OUT_TARGET_CURR_ATT_CLW RSP_SVR_OUT_TARGET_CURR_RATE_CLW	-	$(x\ y\ z\ v_x\ v_y\ v_z$ $q_1\ q_2\ q_3\ q_0\ \omega_x\ \omega_y\ \omega_z)$ q_0 scalar part
Chaser Requested State (U)	RSP_SVR_OUT_CHASER_REQ_POS_U RSP_SVR_OUT_CHASER_REQ_SPEED_U RSP_SVR_OUT_CHASER_REQ_ATT_U RSP_SVR_OUT_CHASER_REQ_RATE_U	-	$(x\ y\ z\ v_x\ v_y\ v_z$ $q_1\ q_2\ q_3\ q_0\ \omega_x\ \omega_y\ \omega_z)$ q_0 scalar part
Target Requested State (U)	RSP_SVR_OUT_TARGET_REQ_POS_U RSP_SVR_OUT_TARGET_REQ_SPEED_U RSP_SVR_OUT_TARGET_REQ_ATT_U RSP_SVR_OUT_TARGET_REQ_RATE_U	-	$(x\ y\ z\ v_x\ v_y\ v_z$ $q_1\ q_2\ q_3\ q_0\ \omega_x\ \omega_y\ \omega_z)$ q_0 scalar part
Chaser Requested State (CLW)	RSP_SVR_OUT_CHASER_REQ_POS_CLW RSP_SVR_OUT_CHASER_REQ_SPEED_CLW RSP_SVR_OUT_CHASER_REQ_ATT_CLW RSP_SVR_OUT_CHASER_REQ_RATE_CLW	-	$(x\ y\ z\ v_x\ v_y\ v_z$ $q_1\ q_2\ q_3\ q_0\ \omega_x\ \omega_y\ \omega_z)$ q_0 scalar part
Target Requested State (CLW)	RSP_SVR_OUT_TARGET_REQ_POS_CLW RSP_SVR_OUT_TARGET_REQ_SPEED_CLW RSP_SVR_OUT_TARGET_REQ_ATT_CLW RSP_SVR_OUT_TARGET_REQ_RATE_CLW	-	$(x\ y\ z\ v_x\ v_y\ v_z$ $q_1\ q_2\ q_3\ q_0\ \omega_x\ \omega_y\ \omega_z)$ q_0 scalar part
Chaser Command Position (CLW)	RSP_SVR_OUT_CHASER_CMD_POS	m	$(x\ y\ z)$
Chaser Command Attitude (CLW)	RSP_SVR_OUT_CHASER_CMD_ATT	-	$(q_1\ q_2\ q_3\ q_0)$, q_0 scalar part
Target Command Position (CLW)	RSP_SVR_OUT_TARGET_CMD_POS	m	$(x\ y\ z)$
Target Command Attitude (CLW)	RSP_SVR_OUT_TARGET_CMD_ATT	-	$(q_1\ q_2\ q_3\ q_0)$, q_0 scalar part
Send 2 Receive	SCP_OUT_SEND2RECV	s	$(\Delta t_{s2r}\ \overline{\Delta t_{s2r}}\ \Delta t_{s2r,noise}$ $\Delta t_{s2r,jitter}\ \Delta t_{s2r,stddev}$ $\Delta t_{s2r,SNR})$
Acquisition 2 Output	SCP_OUT_ACQU2OUTPUT	s	$(\Delta t_{a2o}\ \overline{\Delta t_{a2o}}\ \Delta t_{a2o,noise}$ $\Delta t_{a2o,jitter}\ \Delta t_{a2o,stddev}$ $\Delta t_{a2o,SNR})$

Continued on next page

Table B.2 – *Continued from previous page*

Output	Associated RSP server/SCP Output	Unit	Format
Deadtime in sec	SCP_OUT_DEADTIME_SEC	s	$(\Delta t_{dead}\ \overline{\Delta t_{dead}}\ \Delta t_{dead,noise}\ \Delta t_{dead,jitter}\ \Delta t_{dead,stddev}\ \Delta t_{dead,SNR})$
Deadtime in steps	SCP_OUT_DEADTIME	-	$(\Delta n_{dead}\ \overline{\Delta n_{dead}}\ \Delta n_{dead,noise}\ \Delta n_{dead,jitter}\ \Delta n_{dead,stddev}\ \Delta n_{dead,SNR})$
Connection OK	SCP_OUT_CONNECTION_OK	-	scalar
Client Reference Timestep	SCP_OUT_TARGET_REF_TIMESTEP	-	scalar
Server Reference Timestep	SCP_OUT_HOME_REF_TIMESTEP	-	scalar
Actual Forerun	RSP_SVR_OUT_ACTUAL_FORERUN	-	scalar
Chaser Speed/Acc Violation	RSP_SVR_OUT_CHASER_VIOLATION	-	scalar
Target Speed/Acc Violation	RSP_SVR_OUT_TARGET_VIOLATION	-	scalar
Integrator Enable	RSP_SVR_OUT_INT_ENABLE	-	scalar
Chaser Integrator Force	RSP_SVR_OUT_CHASER_INT_FORCE	N	$(F_x\ F_y\ F_z)$
Chaser Integrator Torque	RSP_SVR_OUT_CHASER_INT_TORQUE	Nm	$(T_x\ T_y\ T_z)$
Target Integrator Force	RSP_SVR_OUT_TARGET_INT_FORCE	N	$(F_x\ F_y\ F_z)$
Target Integrator Torque	RSP_SVR_OUT_TARGET_INT_TORQUE	Nm	$(T_x\ T_y\ T_z)$
Chaser Integrator Initial State	RSP_SVR_OUT_CHASER_INT_INIT_POS_U RSP_SVR_OUT_CHASER_INT_INIT_SPEED_U RSP_SVR_OUT_CHASER_INT_INIT_ATT_U RSP_SVR_OUT_CHASER_INT_INIT_RATE_U	-	$(x\ y\ z\ v_x\ v_y\ v_z\ q_1\ q_2\ q_3\ q_0\ \omega_x\ \omega_y\ \omega_z)$ q_0 scalar part
Target Integrator Initial State	RSP_SVR_OUT_TARGET_INT_INIT_POS_U RSP_SVR_OUT_TARGET_INT_INIT_SPEED_U RSP_SVR_OUT_TARGET_INT_INIT_ATT_U RSP_SVR_OUT_TARGET_INT_INIT_RATE_U	-	$(x\ y\ z\ v_x\ v_y\ v_z\ q_1\ q_2\ q_3\ q_0\ \omega_x\ \omega_y\ \omega_z)$ q_0 scalar part

Table B.3.: Parameters of RemoteSim-Server Simulink Block. A description is given in Tab.2.9 via the associated RSP server parameters and in Tab.2.1 via the associated SCP parameters.

Parameter	Associated RSP server/SCP Parameter	Unit	Format	Example
Port for Control Connection	SCP_PARAM_CONTROL_PORT	-	scalar integer $0-65535$	3000
Port for Data Connection	SCP_PARAM_DATA_PORT	-	scalar integer $0-65535$	3001
IP of Target Machine	SCP_PARAM_IP	-	string	'172.16.247.5'
Connection Timeout	SCP_PARAM_TIMEOUT	ms	scalar integer	4000
Speed Limit	RSP_SVR_PARAM_SPEED_LIMIT	m/s	scalar double	1
Angular Speed Limit	RSP_SVR_PARAM_ANGULAR_SPEED_LIMIT	rad/s	scalar double	$\pi/4$
Acceleration Limit	RSP_SVR_PARAM_ACC_LIMIT	m/s^2	scalar double	0.5

Continued on next page

Table B.3 – *Continued from previous page*

Parameter	Associated RSP server/SCP Parameter	Unit	Format	Example
Angular Acceleration Limit	RSP_SVR_PARAM_ANGULAR_ACC_LIMIT	s	scalar double	$\pi/8$
Chaser Position for Facility Initialization	RSP_SVR_PARAM_CHASER_FACILITY_INIT_POS	m	$(x\,y\,z)$	$[0;0;0]$
Chaser Attitude for Facility Initialization	RSP_SVR_PARAM_CHASER_FACILITY_INIT_ATT	-	$(q_1\,q_2\,q_3\,q_0)$, q_0 scalar part	$[0;0;0;1]$
Target Position for Facility Initialization	RSP_SVR_PARAM_TARGET_FACILITY_INIT_POS	m	$(x\,y\,z)$	$[8;0;0]$
Target Attitude for Facility Initialization	RSP_SVR_PARAM_TARGET_FACILITY_INIT_ATT	-	$(q_1\,q_2\,q_3\,q_0)$, q_0 scalar part	$[0;0;0;1]$
Sampletime	RSP_CLI_PARAM_SAMPLETIME	s	scalar double	0.004
Translational Braking Acceleration	RSP_SVR_PARAM_BRAKE_ACC	m/s^2	scalar double	0.1
Rotational Braking Acceleration	RSP_SVR_PARAM_ANGULAR_BRAKE_ACC	rad/s^2	scalar double	0.1

C. Software Documentation

C.1. Overall class Structure

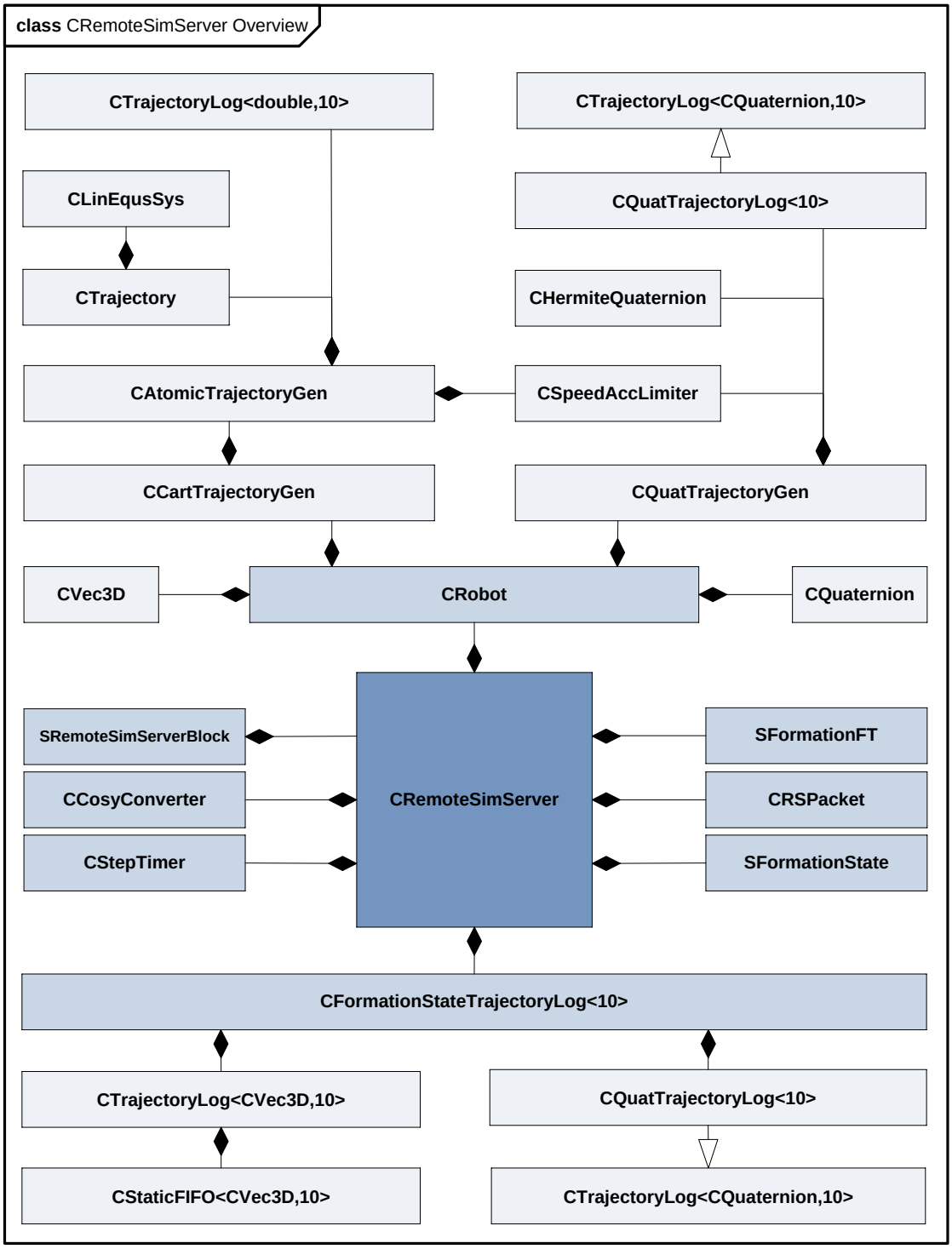

Figure C.1.: Overview of class CRemoteSimServer

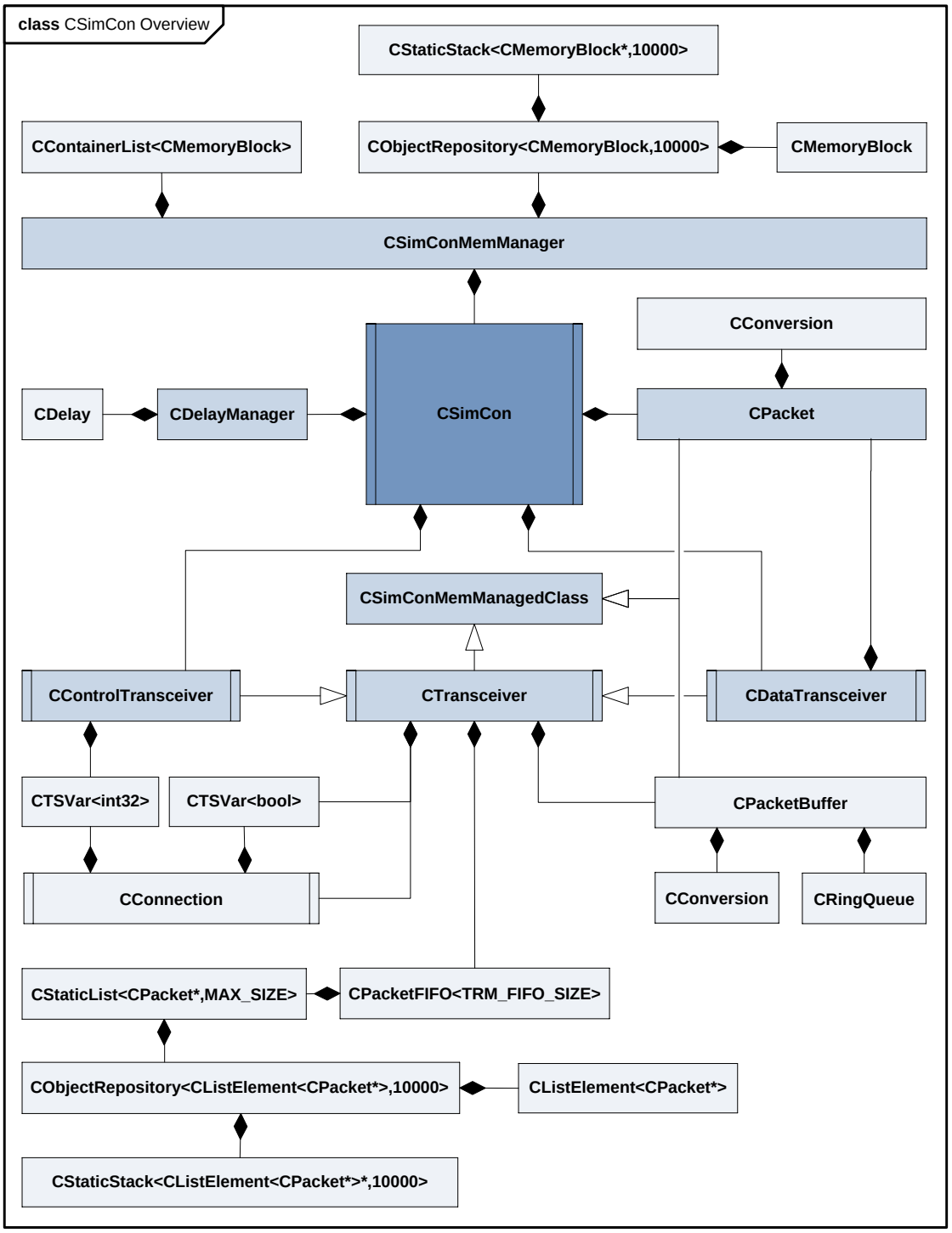

Figure C.2.: Overview of class CSimCon

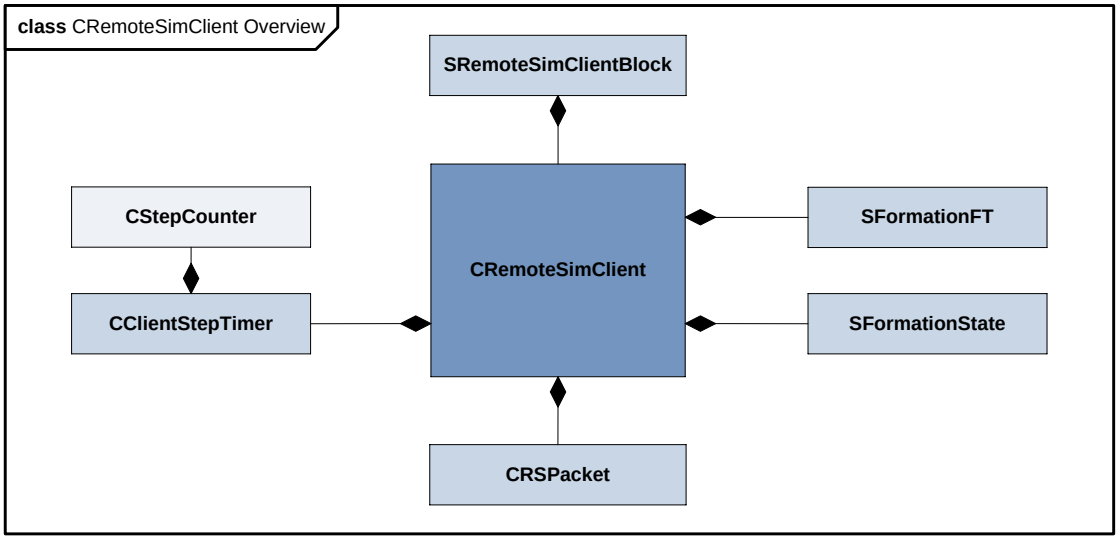

Figure C.3.: Overview of class CRemoteSimClient

C.2. class CAtomicTrajectoryGen

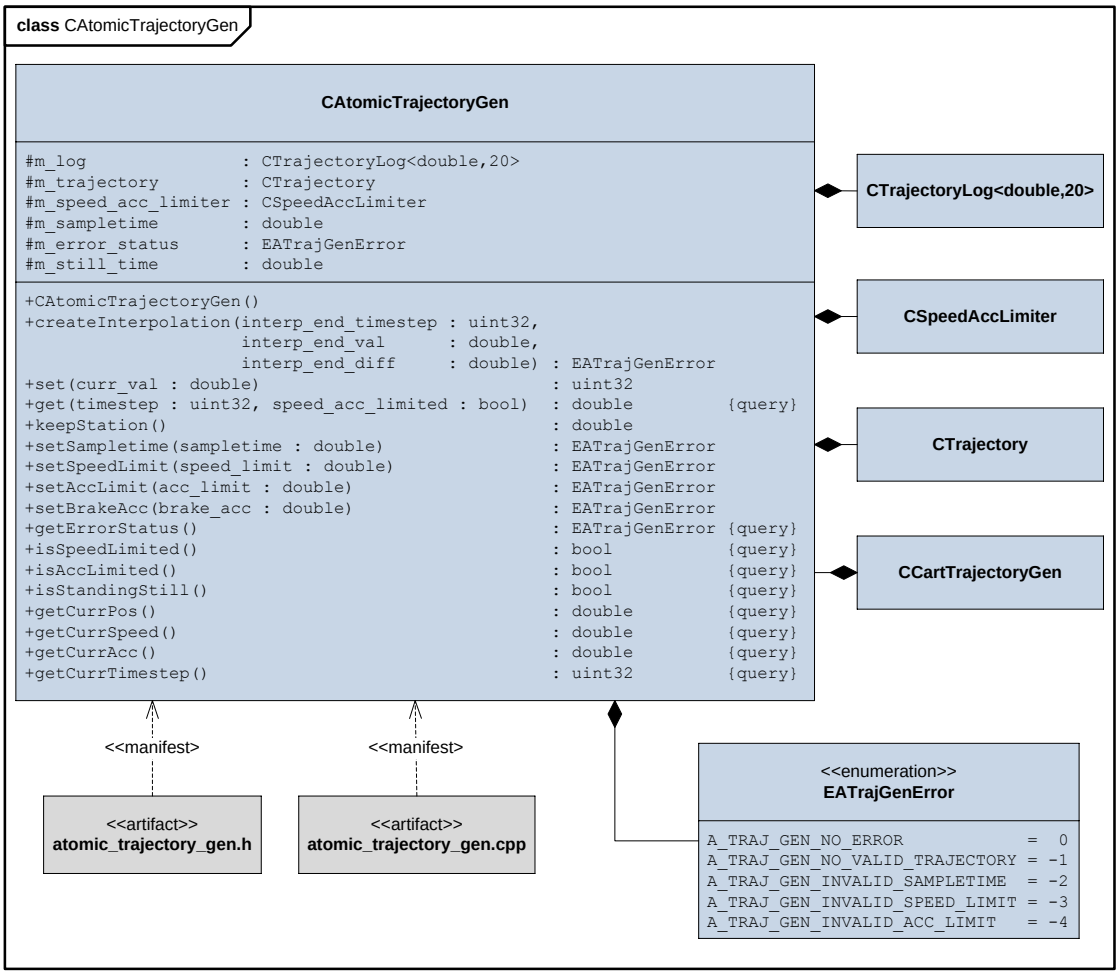

Figure C.4.: UML Class Diagram of class CAtomicTrajectoryGen

Description: The `class CAtomicTrajectoryGen` generates a 1D C^1 continuous translational trajectory with the ability to limit speed and acceleration. The `class` takes care about calculation of speed, so that merely position has to be provided at each time step. First, the methods `setSampletime(...)`, `setSpeedLimit(...)`, `setAccLimit(...)` and `setBrakeAcc(...)` have to be executed to set the associated parameters. Then, at each time step `set(...)` has to be called with the actual position as argument. (Assuming that the robots move ideally, this may be the commanded value of the last time step.) To obtain trajectory values, the method `get(...)` has to be used with the time step of the desired trajectory value and a flag indicating whether to limit robots' speed and acceleration as arguments. The return value of `get(...)` depends on the state `CAtomicTrajectoryGen` is in, e.g. whether the method `crealteInterpolation(...)` or `keepStation(...)` has been executed last. In the former case, values of the interpolated trajectory - based on end time step, end position and end speed provided to `createInterpolation(...)` - are returned. In the latter case, the robots are decelerated to zero speed according to the algorithm presented in Sec.2.6.4.3. Interpolation is handled by the member variable `m_trajectory`, see Sec.C.39. Keep station is handled by the member variable `m_speed_acc_limiter`, see Sec.C.33. `isSpeedLimited()` and `isAccLimited()` indicate that the robots' commands would have led to violations of speed and/or acceleration and that therefore the robots' movements has been modified to comply with the limitations. `isStandingStill()` returns true if the robots' speed has been zero for a certain amount of time, specified as constant `STILL_TIMEOUT` in file atomic_trajectory_gen.h. `getCurrPos()`, `getCurrSpeed()` and `getCurrAcc()` return current position, speed and acceleration, calculated from the values given via `set(...)`. The member variable `m_log` is used for that

purpose, see Sec.C.40. In general, `getCurrPos()` does not return the same value as `get(...)` with the current time step as argument. Current time step can be obtained by `getCurrTimestep()`. Overall, the process is comprised of setting the current position via `set(...)` and retrieving the next position via `get(...)` at each time step. At any point, either `createInterpolation(...)` or `keepStation()` are executed to determine the output behaviour of `CAtomicTrajectoryGen`, which stays that way until one of these methods is called again (i.e. resulting in a new interpolated trajectory or changing from interpolation to keep station).

Context: Three instances of this `class` are part of `CCartTrajectoryGen` to realize the 3D translational trajectory of a robot.

C.3. class CCartTrajectoryGen

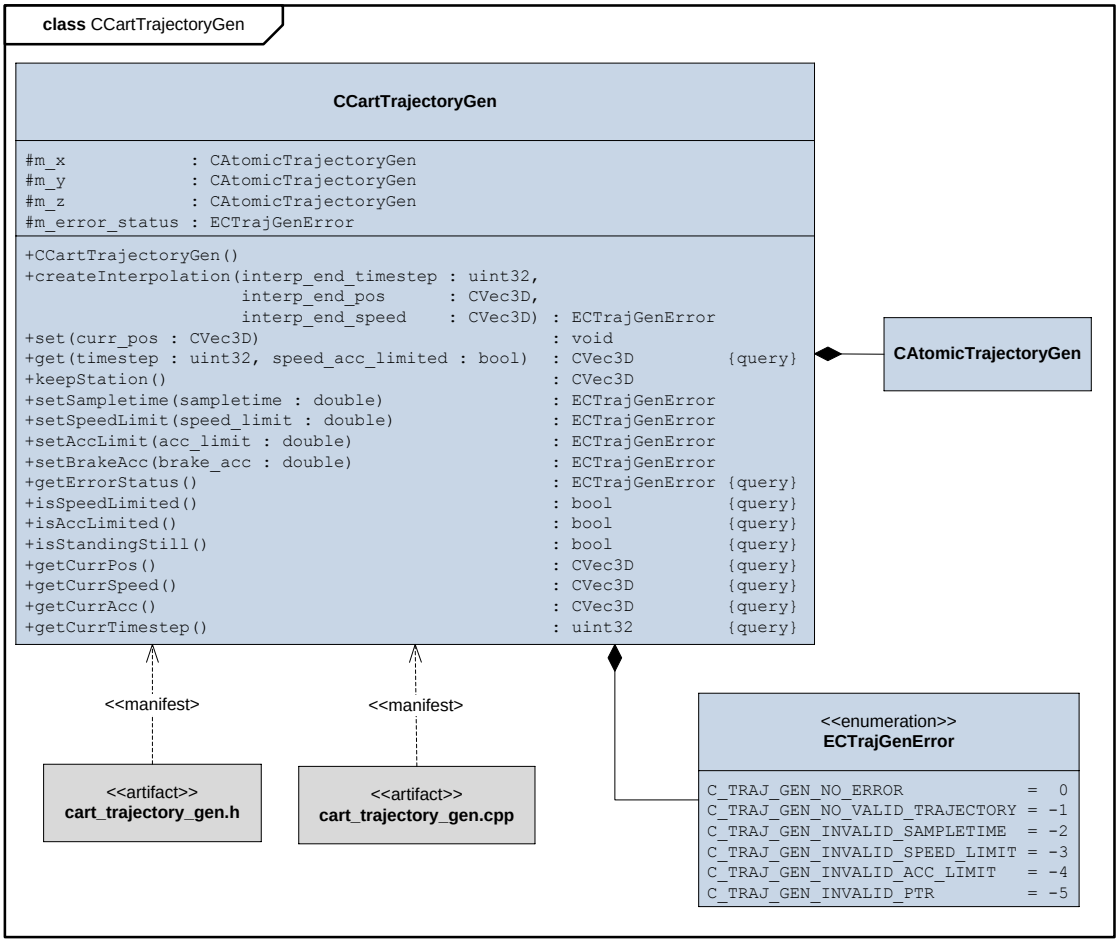

Figure C.5.: UML Class Diagram of class CCartTrajectoryGen

Description: The `class` `CCartTrajectoryGen` combines three instances of `CAtomicTrajectoryGen` (`m_x`, `m_y` and `m_z`), each generating a 1D trajectory, to realize a full three-dimensional position trajectory generator thereby including all functionalities of `CAtomicTrajectoryGen`. In fact, the public methods of `CCartTrajectoryGen` are similar to those of `CAtomicTrajectoryGen` with one-dimensional trajectory values replaced by a vector in the form of `class CVec3D`. Therefore, methods are not described further here. The reader may refer to Sec.C.2.

Context: An instance of `CCartTrajectoryGen` is part of `CRobot`. It deals with the translational part of the robot trajectory.

C.4. class CClientStepTimer

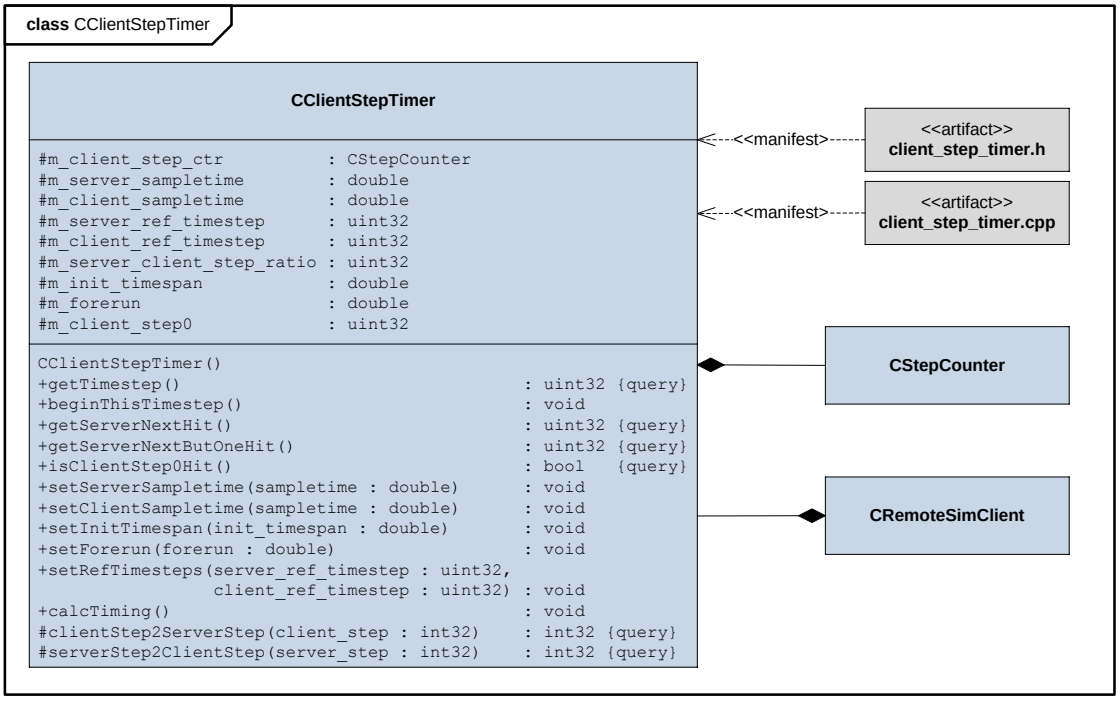

Figure C.6.: UML Class Diagram of class CClientStepTimer

Description: `CClientStepTimer` takes care about all timing tasks RemoteSim-Client has to carry out and relies on. A number of functions have to be called as initialization in order to set client and server sample time, reference time steps, initial timespan (for initial trajectory) as well as desired forerun. At the beginning of each simulation time step, the method `beginThisTimestep()` has to be called, which merely executes `m_client_step_ctr++`. The current time step can be obtained by `getTimestep()`. With all initial parameters set, the method `calcTiming()` carries out the algorithm comprised of (2.19), (2.20) and (2.21). Then, server next hit `getNextHit()` and server next-but-one hit `getNextButOneHit()` can be returned any time. Moreover, the method `isClientStep0Hit` signals, when RemoteSim-Client is to transit from `STATE_SIM_INIT` to `STATE_SIM_TRAJECTORY`/`STATE_SIM_FORCE_TORQUE`. In addition to these public methods, there are also two protected methods. `clientStep2ServerStep(...)` realizes (2.11) and `serverStep2ClientStep(...)` the inverse.

Context: An instance of `CClientStepTimer` is part of `CRemoteSimClient`.

C.5. class CConnection

Description: The `class CConnection` is an active `class`. It is initialized (`createConnection(...)`) with IP address, port number and type (TCP server or client). Thereafter, `CConnection` starts a thread `connectionThreadServer()` or `connectionThreadClient()`, which act as a TCP server waiting for a client or as a client trying to contect to a server. As soon as a connection is established, the connection thread is terminated in the case of a client or goes to idle mode in the case of the server. The method `getSocket()` returns the socket that can be used to transmit and receive data. The method `isConnected()` returns `true` as soon as the connection is established. It does not change to `false` again, if the connection is not intact any more. While `closeConnection()` closes the link, the method `reconnect()` closes the connection and trys to reconnect.

Context: An instance of this `class` is part of `CTransceiver`, where it handles establishement of a TCP/IP connection and provides an according socket.

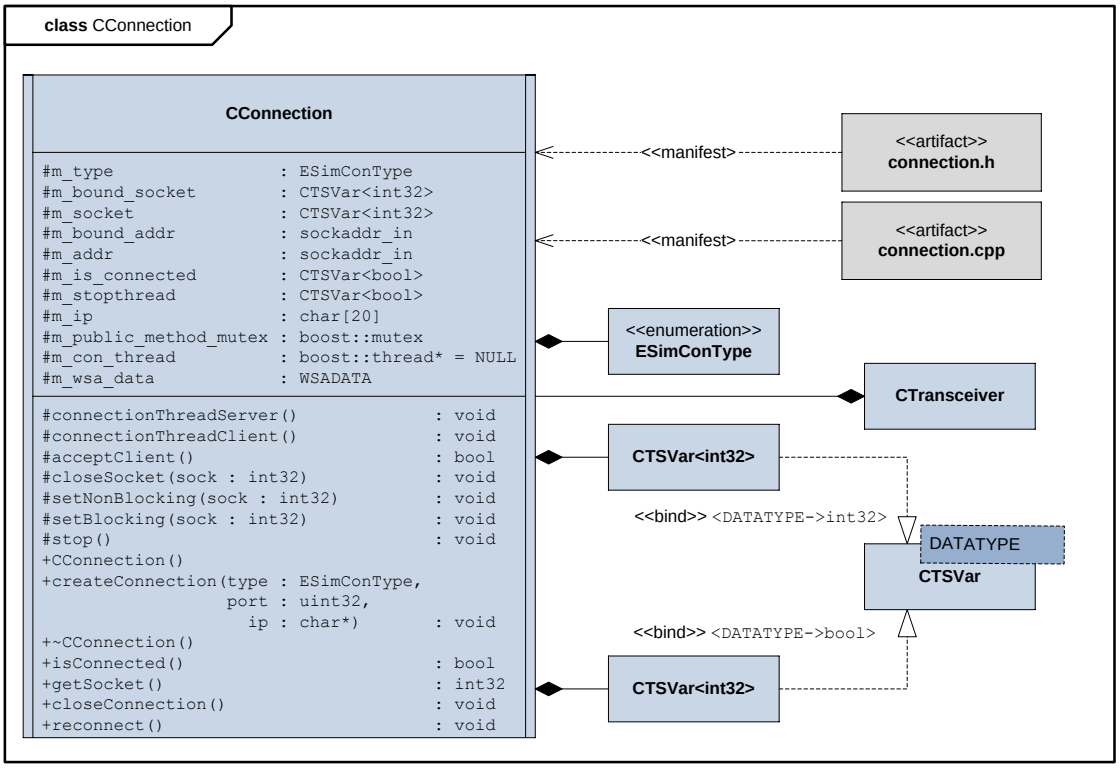

Figure C.7.: UML Class Diagram of class CConnection

C.6. class CConversion

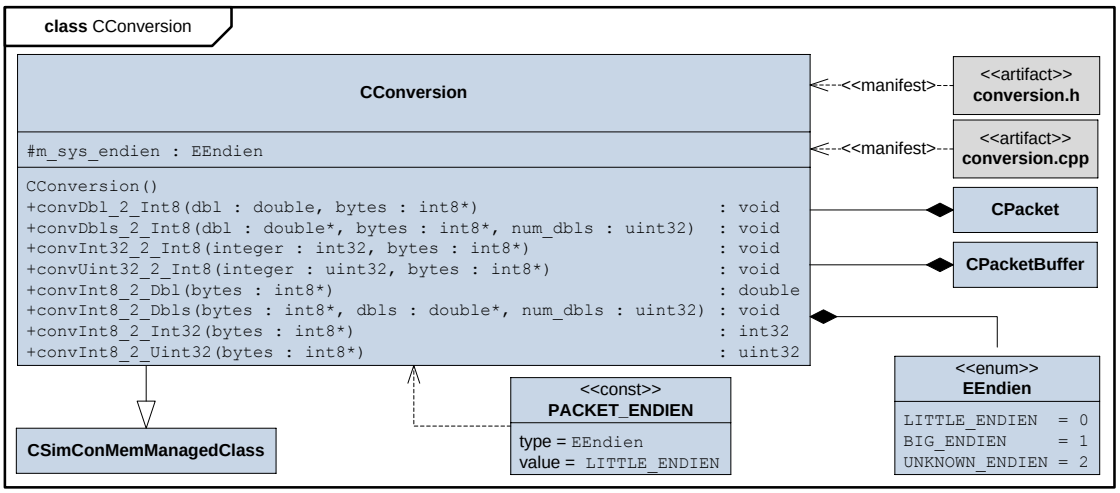

Figure C.8.: UML Class Diagram of class CConversion

Description: The `class CConversion` is used with `CPacket` and `CPacketBuffer`. SCP packets are transmitted via TCP as a stream of bytes. Integers and doubles are comprised of several bytes. The order of these bytes in memory (endianness) depends on the platform. The array of bytes in an SCP packet follows little endian format (determined by constant `PACKET_ENDIEN`). `CConversion` converts integers and doubles to an array of bytes consistent with the platforms endianness. For that purpose, `CConversion` determines the endianness upon instanciation by reading a characteristic

integer from memory byte by byte. Thereupon, the member variable `m_sys_endien` contains the platform's endianness. All methods are conversion functions and are self-explanatory.

Context: An instance of the `class` is part of `CPacket` and `CPacketBuffer`.

C.7. class CCosyConverter

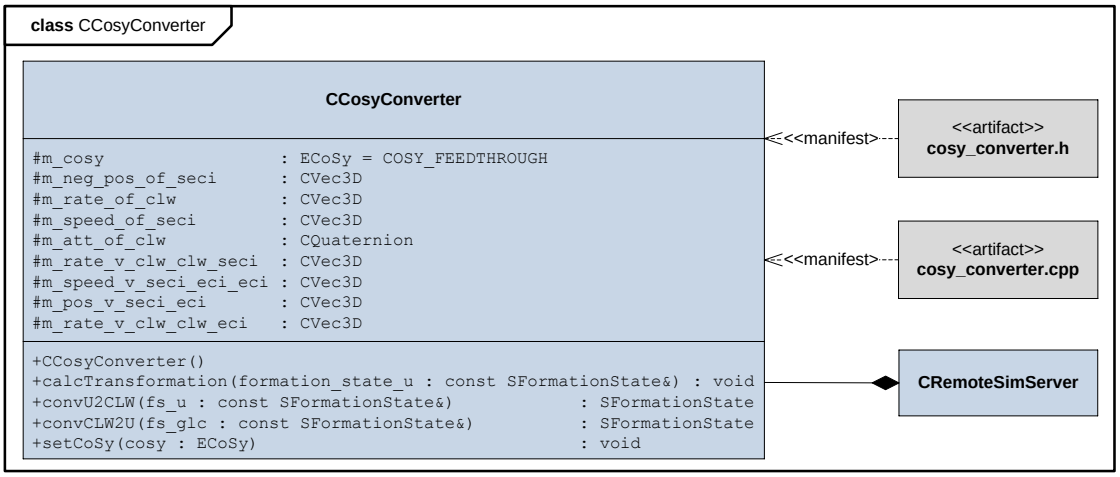

Figure C.9.: UML Class Diagram of class CCosyConverter

Description: The `class` `CCosyConverter` implements the coordinate transformation algorithm outlined in Sec.2.6.4.6. By default, the `class` realizes `COSY_FEEDTHROUGH`. With `setCoSy(...)` the type of coordinate system can be set, i.e. `COSY_ECI_2_CLW`. Calling the method `calcTransformation` has the `class` determine the transformation. Then `convU2CLW(...)` and `convCLW2U(...)` can be used to carry out the transformation from U to CLW or from CLW to U frame.

Description: An instance of `CCosyConverter` is part of `CRemoteSimServer`.

C.8. class CContainerList

Description: The `template` `CContainerList` realizes a standard list where the list elements have to be provided to the `template` "from outside". The `template` argument determines the type of list elements. It can be any type with the following restrictions: It must provide the methods `getNext()`, `getPrevious()`, `setNext(...)` and `setPrevious(...)`, like i.e. `CMemoryBlock`. In contrast to the list of the C++ standard library, methods like `insert(...)` and `remove(...)` deal with the list elements itself and not with the values the list elements contain. This allows to access the list, if the address of a certain list element is known. This makes sense, if the list elements are not only part of an instance of `CContainerList`, but are also linked to another data structure, i.e. for sorting according to certain attributes of the list elements. This is exploited with `CSimConMemManager`, where an instance of `CContainerList` represents the sequence of memory blocks in memory, while direct access to the list elements is used to manage free and used memory blocks.

Context: An instance of `CContainerList` is part of `CSimConMemManager` in connection with the list elements `CMemoryBlock`.

C.9. class CControlTransceiver

Description: This `class` implements the SCP control connection. The method `init(...)` must be used to initialize the `class` with the required SCP parameters. `pingThread()` is executed as a separate thread which sends ping packets in regular intervals and checks if a ping packet has been mirrored successfully by the communication partner. Then, `connectionOK()` indicates whether a connection is intact or not. `reftimeThread()` is executed as a separate thread and sends delay

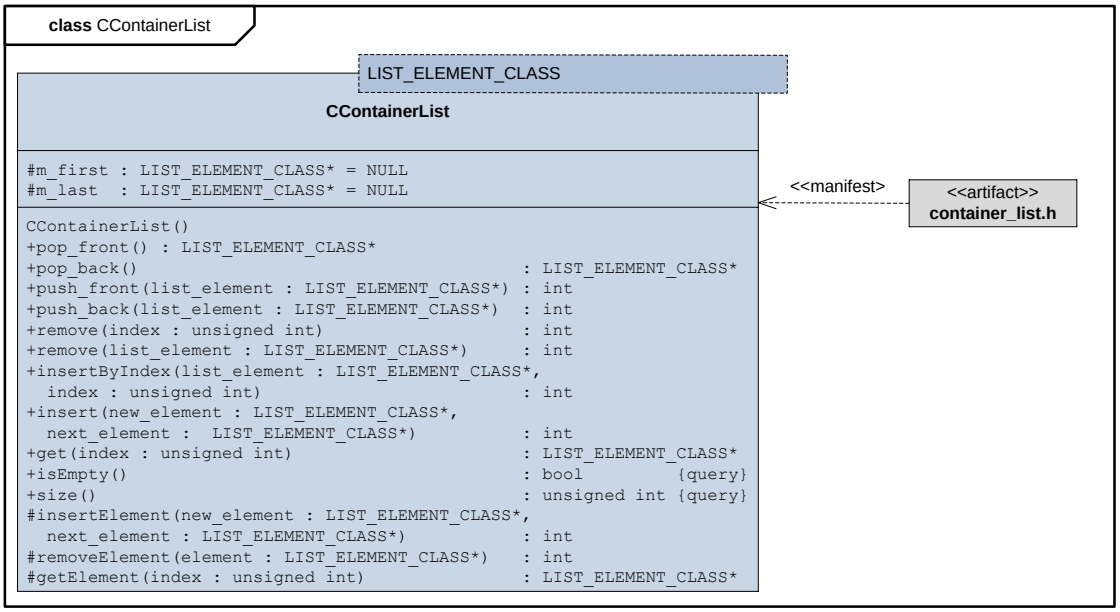

Figure C.10.: UML Class Diagram of class CContainerList

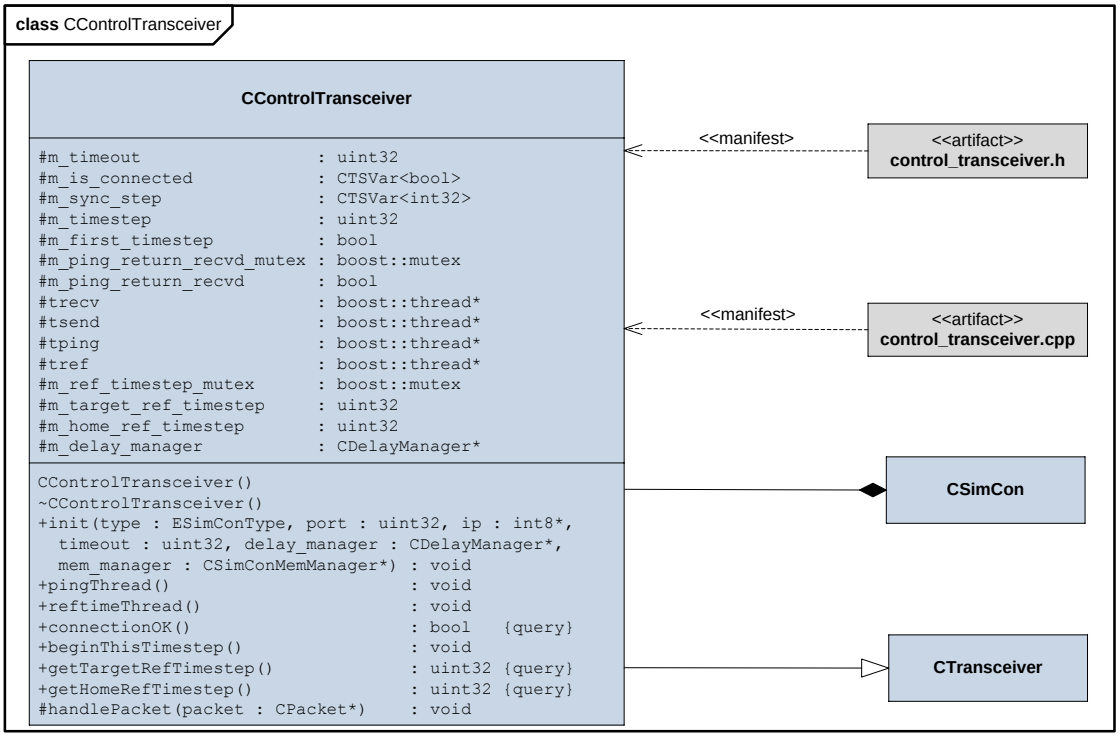

Figure C.11.: UML Class Diagram of class CControlTransceiver

packets in regular intervals. `beginThisTimestep()` has to be executed at the beginning of each time step. This logs time steps and sends a ref packet to determine reference time steps. These can be obtained by `getTargetRefTimestep()` and `getHomeRefTimestep()`.

Context: An instance of `CControlTransceiver` is part of `CSimCon`.

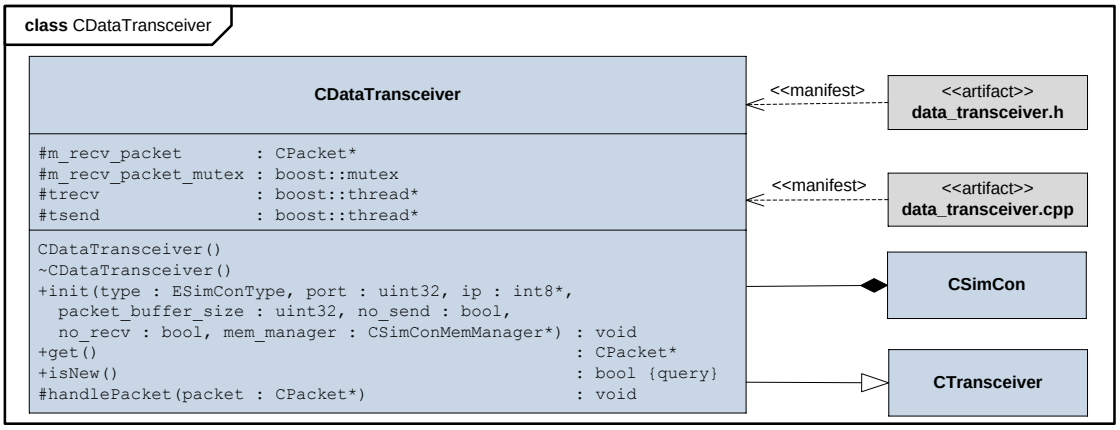

Figure C.12.: UML Class Diagram of class CDataTransceiver

C.10. class CDataTransceiver

Description: The `class CDataTransceiver` implements the SCP data connection. It makes only minor extensions to `CTransceiver`. Obligatorily, the method `init(...)` must be used to provide the `class` with the required SCP parameters. `isNew()` indicates that a data packet is available that has not been read yet. `get()` returns a received SCP packet (`CPacket`). Methods for transmitting packets are inherited from `CTransceiver`.

Context: An instance of `CDataTransceiver` is part of `CSimCon`.

C.11. class CDelay

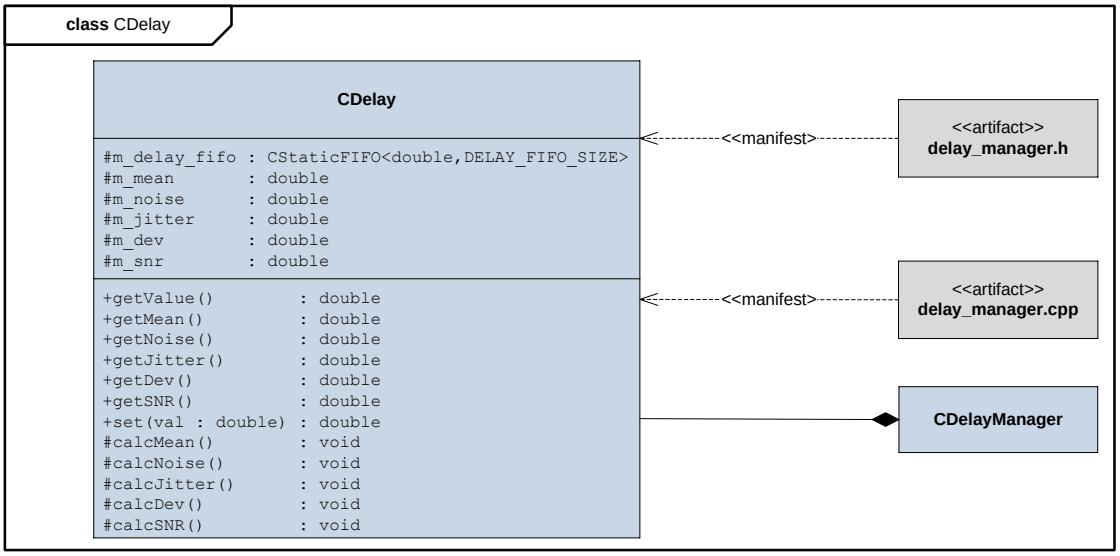

Figure C.13.: UML Class Diagram of class CDelay

Description: The `class CDelay` buffers packet delay values, i.e. send to receive, in a FIFO (member variable `m_delay_value`) by a call to `set(...)`. It provides methods for obtaining the value (`getValue()`), mean (`getMean()`), noise (`getNoise()`), jitter (`getJitter()`), standard deviation (`getDev()`) and Signal to Noise Ratio (`getSNR()`) from the buffered sequence of delay values. Protected member methods are used to calculate these characteristic values, with n being the current time step:

`calcMean()`

$$(\overline{\Delta t})_n = \frac{\sum_{j=n-19}^{n} (\Delta t)_j}{20} \tag{C.1}$$

`calcNoise()`

$$(\Delta t_{noise})_n = (\Delta t)_n - (\overline{\Delta t})_n \tag{C.2}$$

`calcJitter()`

$$(\Delta t_{jitter})_n = (\Delta t)_n - (\Delta t)_{n-1} \tag{C.3}$$

`calcDev()`

$$(\Delta t_{stddev})_n = \sqrt{\frac{\sum_{j=n-19}^{n} \left[(\Delta t)_j - (\overline{\Delta t})_n\right]^2}{20}} \tag{C.4}$$

`calcSNR()`

$$(\Delta t_{SNR})_n = \frac{(\overline{\Delta t})_n}{(\Delta t_{stddev})_n} \tag{C.5}$$

Context: For each of the delays SCP defines, `CDelayManager` contains an instance of `CDelay`.

C.12. class CDelayManager

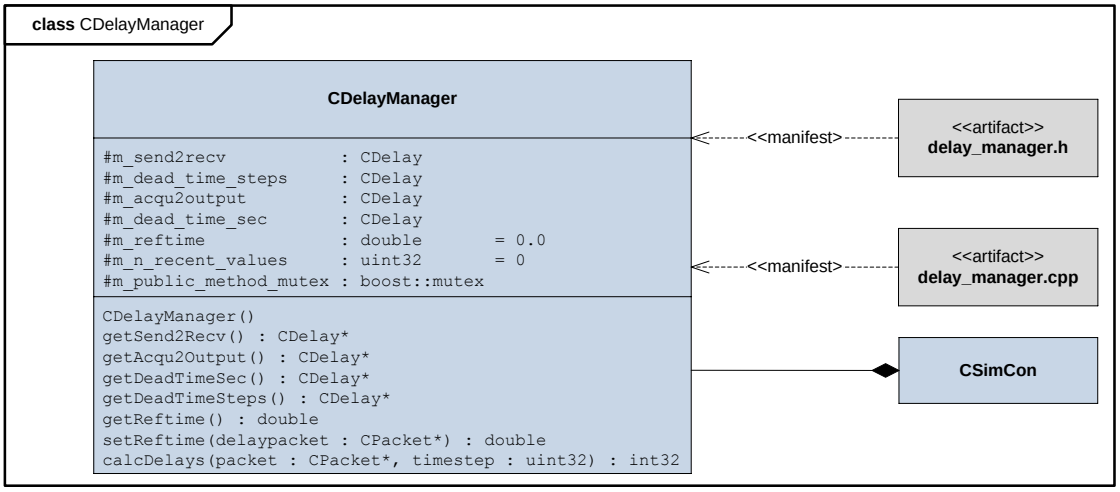

Figure C.14.: UML Class Diagram of class CDelayManager

Description: The `class CDelayManager` reads a SCP packets header fields (method `calcDelays(...)`) and determines the packet delays SCP defines, thus implementing (2.7), (2.8),(2.9) and (2.10). Before, the reference time must be set by calling `setReftime(...)` using a received delay packet. Each of the delays is buffered in an instance of `CDelay`. Pointers to these instances can be obtained by the associated get... methods. Thereupon, the methods of `CDelay` can be used to get the delay's characteristical values like mean, noise...

Context: An instance of `CDelayManager` is part of `CSimCon`.

C.13. class CHermiteQuaternion

Description: The `class CHermiteQuaternion` realizes the quaternion interpolation algorithm presented in Sec.2.6.4.5. The protected method `cumulativeBernsteinBasis(...)` implements Eq.(2.40) and `bezierQuaternionCurve(...)` implements Eq.(2.45). To compute a new curve, `create(...)` is called with start time, attitude and angular velocity as well as end time, attitude

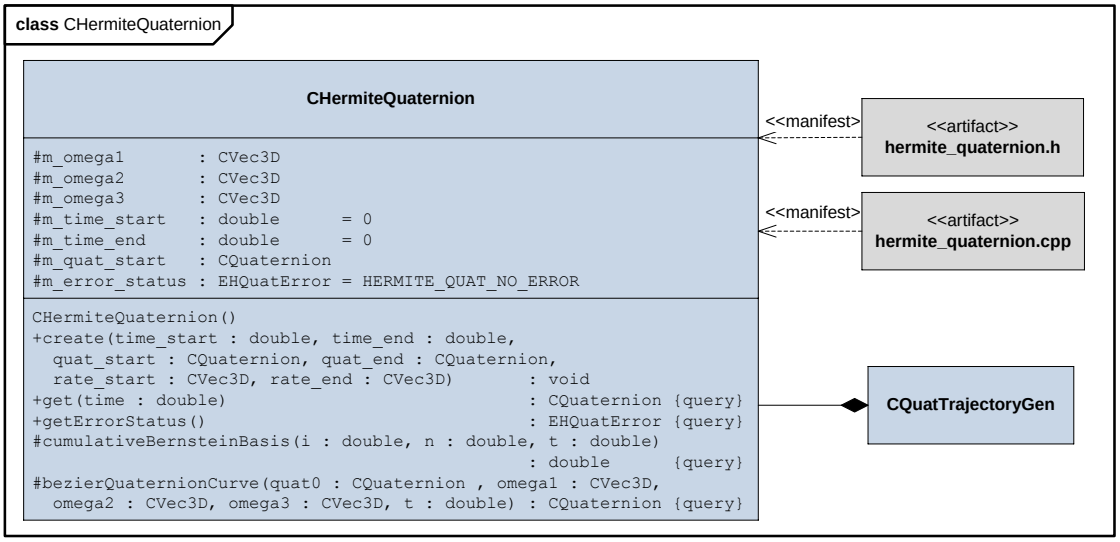

Figure C.15.: UML Class Diagram of class CHermiteQuaternion

and angular velocity as arguments. This method carries out the calculations given by (2.46), (2.47) and (2.48). After these calculations, `get(...)` returns the attitude related to the time supplied as argument. In a way, `CHermiteQuaternion` is the attitude equivalence to `CTrajectory`.

Context: An instance of `CHermiteQuaternion` is part of `CQuatTrajectoryGen` and handles quaternion interpolation.

C.14. class CLinEqusSys

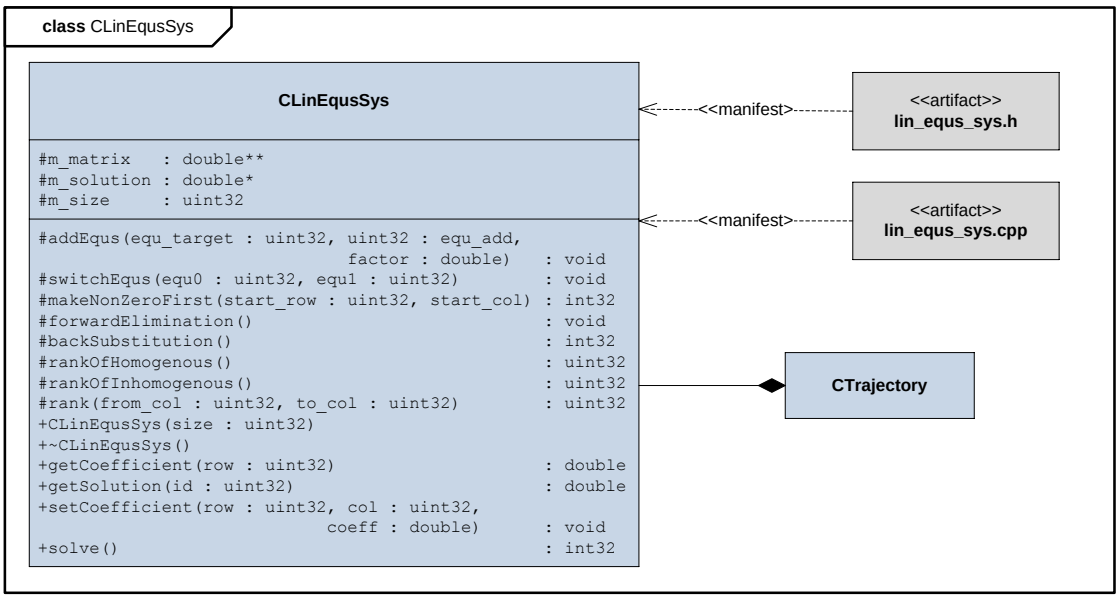

Figure C.16.: UML Class Diagram of class CLinEqusSys

Description: The `class CLinEqusSys` represents a system of linear equations which can be solved by the Gauß algorithm (see [15, p.92]). The number of unknowns must equal the number of equations. The member variable `m_matrix` is a 2D array and stores the coefficient matrix of the equation system. This matrix is square. `m_solution` represents the solution vector and `m_size` the

number of unknowns, e.g. the size of the square coefficient matrix and the size of the solution vector. Several protected methods carry out elementary row operations as part of the algorithm. `addEqus(...)` adds one eqation (matrix row) multiplied by a scalar to another equation (matrix row). `switchEqus(...)` swaps two equations (matrix rows). `makeNonZeroFirst(...)` makes use of `switchEqus(...)` to exchanges equations such, that the element specified as argument to `makeNonZeroFirst(...)` is different from zero. Thus the equations can be ordered as required for the Gauß algorithm. The method `forwardElimination()` transforms the system to triangular form. And provided that the system has that form, `backSubstitution()` finally solves the system whereupon the vector `m_solution` contains the result. The elements of this vector can be obtained by calling `getSolution(...)`. `rankOfHomogenous()`, `rankOfInhomogenous()` and `rank(...)` are used to assess if the system can be solved at all. The public member functions use all of these protected methods. The `class` is initialized by the constructor with system size as an argument. `setCoefficient(...)` must be used to set the individual coefficients of the equation system. Assume a 2x2 system: The coefficient matrix is given by rows 0 to 1 and columns 0 to 1. The right side is column 2, rows 0 to 1. So for a 2x2 system, 6 coefficients must be set. Then execution of `solve()` has the `class` calculate the solution which can be read, element by element, via `getSolution(...)`.

Context: An instance of `CLinEqusSys` is part of the `class CTrajectory`, representing the linear equation system as part of translational trajectory interpolation.

C.15. class CListElement

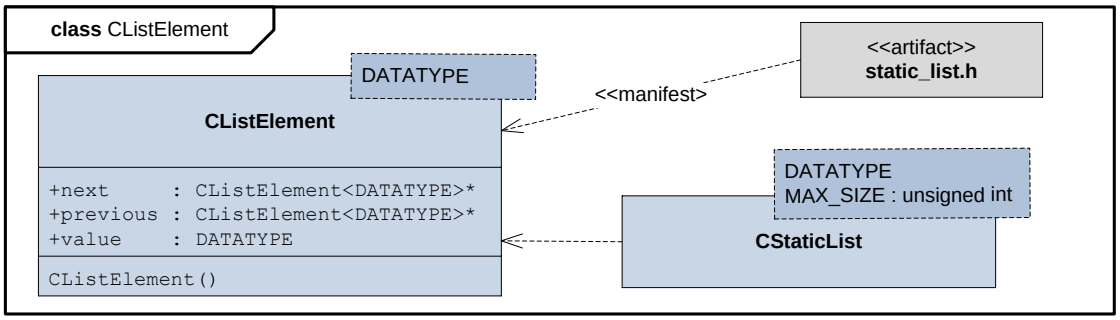

Figure C.17.: UML Class Diagram of class CListElement

Description: This very simple `template` is used with `CStaticList`. It constitutes the elements of the list. Each one contains a pointer to the next list element and one to the previous list element. Another value is the value of the list element. Its type is given as template argument. Save for the constructor, there are no methods. All member variables are public and can be accessed directly.

Context: The list realized by `template CStaticList` is comprised of instances of the `template CListElement`.

C.16. class CMemoryBlock

Description: The `CMemoryBlock` represents a contiguous block of memory. This block can be free or in use, which can be determined by method `isFree()` and changed by `setFree()` and `setNotFree()`. Moreover, a the starting address of the block can be set by `setBlockAddr(...)` and obtained by `getBlockAddr()`. (This is not the address of the intance of `CMemoryBlock`.) Similarly, the length of memory represented by the instance of `CMemoryBlock` can be set (`setBlockLength(...)`) and (`getBlockLength()`). Moreover, the `class` provides the methods that make it compatible with `CContainerList`.

Context: `CMemoryBlock` is used by `CSimConMemManager` in connection with `CContainerList`.

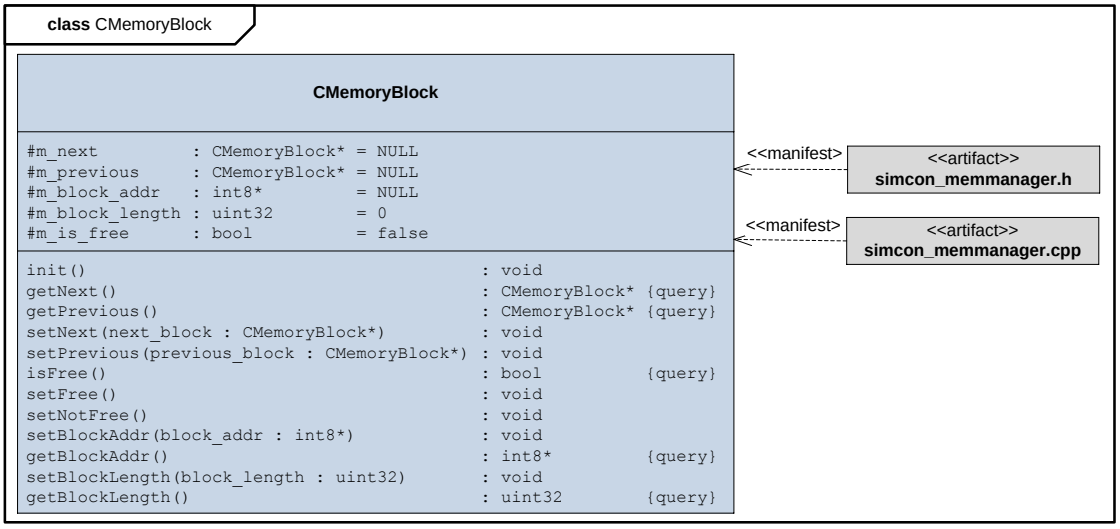

Figure C.18.: UML Class Diagram of class CMemoryBlock

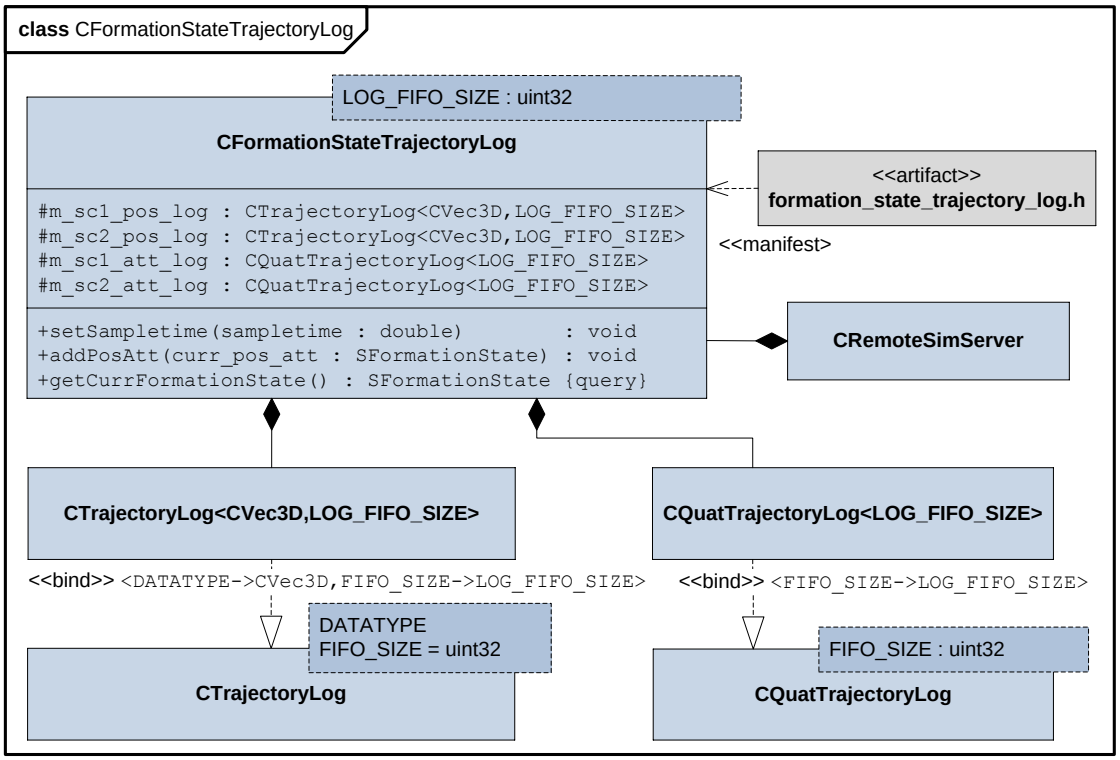

Figure C.19.: UML Class Diagram of class CFormationStateTrajectoryLog

C.17. class CFormationStateTrajectoryLog

Description: The `template CFormationStateTrajectoryLog` combines two instances of each `template CTrajectoryLog` and `template CQuatTrajectoryLog`. Thus, the states of two spacecraft can be handled at once. Before usage, the simulation's sample time has to be set via method `setSampletime()`. Then, `addPosAtt(curr_pos_att : SFormationState)` adds the formation state (states of two spacecraft) to the FIFO, demanding a `struct SFormationState` as argument. Note that only the `pos` and `att` fields of the `struct` are considered for this method. `speed` and `rate` are then calculated from the recently added states and are included in the return value (of type `SFormationState` of method `getCurrFormationState()`.

Context: An instance of `CFormationStateTrajectoryLog` is used by `CRemoteSimServer` for logging the current positions and attitudes of the robots as returned by the EPOS CMD interface. Speed and angular velocity are then calculated and can be displayed in addition to position and attitude.

C.18. class CObjectRepository

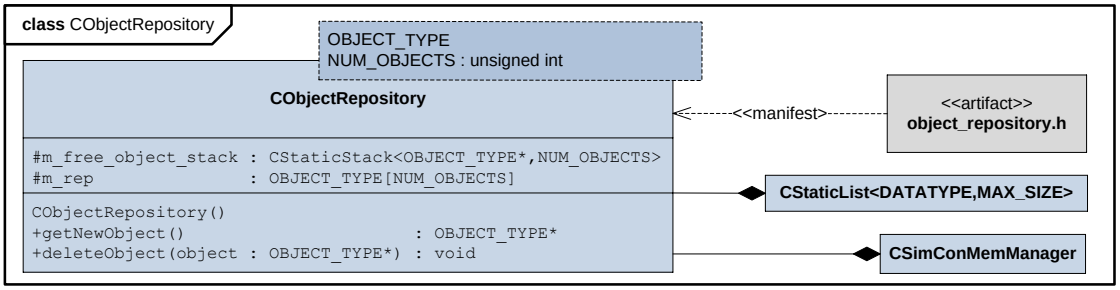

Figure C.20.: UML Class Diagram of class CObjectRepository

Description: This `template` provides a number of objects, whereby static memory (stack) is used, rather than dynamic memory (heap). The first `template` determines type and the second the number of objects the repository can provide. `getNewObject()` returns a pointer to a new object of the repository. `deleteObject(...)` marks the object as not used any more, whereupon it can be obtained as a new object again.

Context: To avoid dynamic memory allocation with VxWorks, a custom memory manager is used `CSimConMemManager`. To manage memory blocks, this memory manager needs a variable number of instances of `CMemoryBlock`. It would be absurd to allocate them from dynamic memory. To realize a repository for these blocks, the `template CObjectRepository` is used.

C.19. class CPacket

Description: The `class CPacket` implements the SCP packet. Its public member methods comprise `get...` and `set...` functions for all header fields. Moreover, the header fields corresponding to times can be copied from another packet (`copyHeaderTimes(...)`) and the fields with index 0 can be copied to those with index 1 (`copyHeaderTimes(...)`. The data pointer of the byte array that constitutes the packet can be accessed directly via `getDataPtr()` for fast access. If the packet is a data packet, then data values can be set and read with the according methods. In general, individual values can be set by index (i.e. `setDblValueByIndex(...)`) or an whole array of specified length can be set at once (i.e. `setDblValues(...)`). Of course, there are according functions for obtaining the data. Several protected methods support execution of the public methods in order to avoid redundancy. It is important to note, that values are always put into the byte array of the packet in little endian format, no matter the endianness of the platform. This is carried out by an instance of `CConversion` and makes the SCP packet format platform independent.

Context: `CPacket` is used with SimCon at various points as SCP packets.

C.20. class CPacketBuffer

Description: The `class CPacketBuffer` buffers the stream of bytes received via TCP/IP with its member variable `m_ring_queue` of `class CRingQueue`. `addData(...)` only puts the bytes into the ring queue, while `addDataAndWritePacket(...)` adds data and returns a SCP packet, if one is contained in the ring queue, e.g. in the received stream of bytes. `writePacket(...)` behaves similarly, except that no data is put into the ring queue. For checking the stream for packets, the protected method `isValidPacketData(...)` examines a portion of the ring queue, starting with

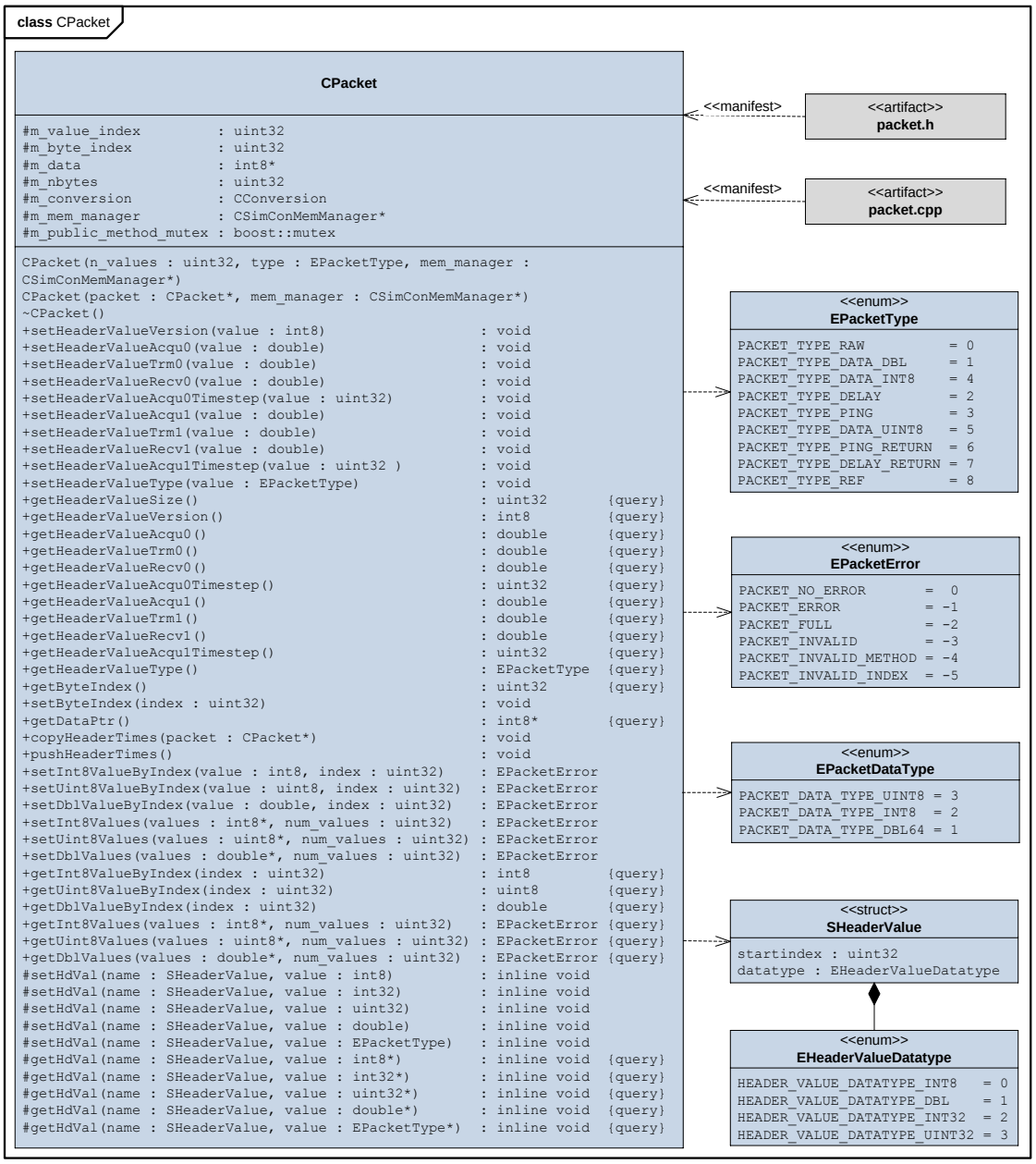

Figure C.21.: UML Class Diagram of class CPacket

the index given as argument, for a valid SCP packet. Packets are recognised by start bytes, length and end bytes.

Context: An instance of `CPacketBuffer` is part of `CSimCon`.

C.21. class CPacketFIFO

Description: The `class CPacketFIFO` enqueues instances of `CPacket` to be transmitted by `CTransceiver` (e.g. `CControlTransceiver` and `CDataTransceiver`). There are several ways of adding a packet to the queue. `add(...)` adds a packet to the queue, where the argument determines the transmission priority. Packets added with a high priority are sent next. Packets added with a low priority are sent after all other packets in the FIFO have been transmitted. `addExclusive(...)` adds a packet to the queue and removes all other packets of the same type. This is used with data

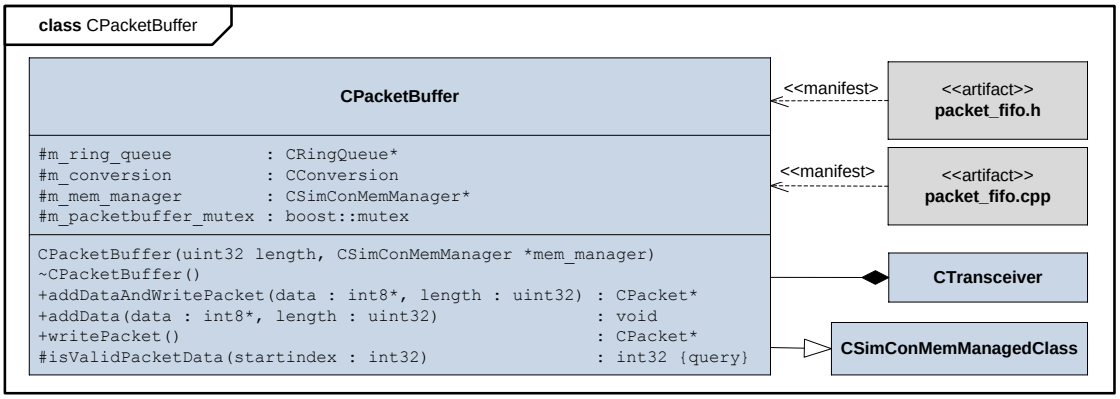

Figure C.22.: UML Class Diagram of class CPacketBuffer

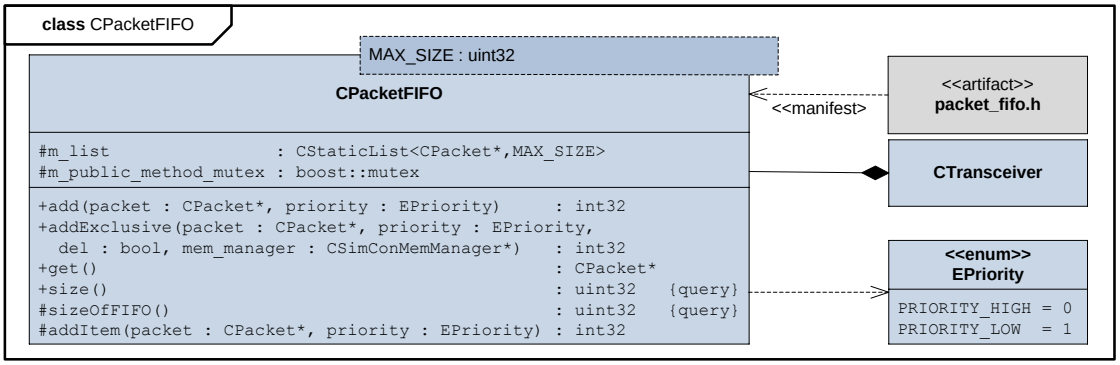

Figure C.23.: UML class diagram of class CPacketFIFO

packets, to ensure that always the latest obtained data is transmitted. The method `get(...)` returns the next packet.

Context: An instance of `CPacketFIFO` is part of `CTransceiver`.

C.22. class CQuaternion

Description: The `class CQuaternion` implements the data type quaternion with the associated operators. Most of the methods are self-explanatory. Therefore, only those which require further explanations are treated here. The constructor `CQuaternion(angle : double, axis : CVec3D)` creates an unit quaternion according to (1.1). The constructor `CQuaternion(vec : CVec3D)` creates a vector quaternion from a vector according to (1.4). The constructor `CQuaternion(unit_vec_x : CVec3D, unit_vec_y : CVec3D, unit_vec_z : CVec3D)` creates a transformation quaternion from the unit coordinate axis of the new frame as illustrated in Sec.2.6.4.6. The method `trans(...)` implements the coordinate transformation (1.2) and the method `revTrans(...)` implements the reverse transformation. Note that all binary functions have to be implemented as separate functions and not as member methods. `vec2quat(..)` returns a vector quaternion from a vector and `quat2vec(..)` returns a vector from a vector quaternion.

Context: This `class` is used at numerous points as attitude and its derivatives.

C.23. class CQuatTrajectoryGen

Description: The `class CQuatTrajectoryGen` can be considered to be the quaternion equivalence of `CCartTrajectoryGen`, with position (`CVec3D`) replaced by attitude (`CQuaternion`) and speed (`CVec3D`) replaced by angular velocity (`CVec3D`). For quaternion interpolation, an instance of `CHermiteQuaternion` is used. Speed/acceleration limitation and keep station functionality are real-

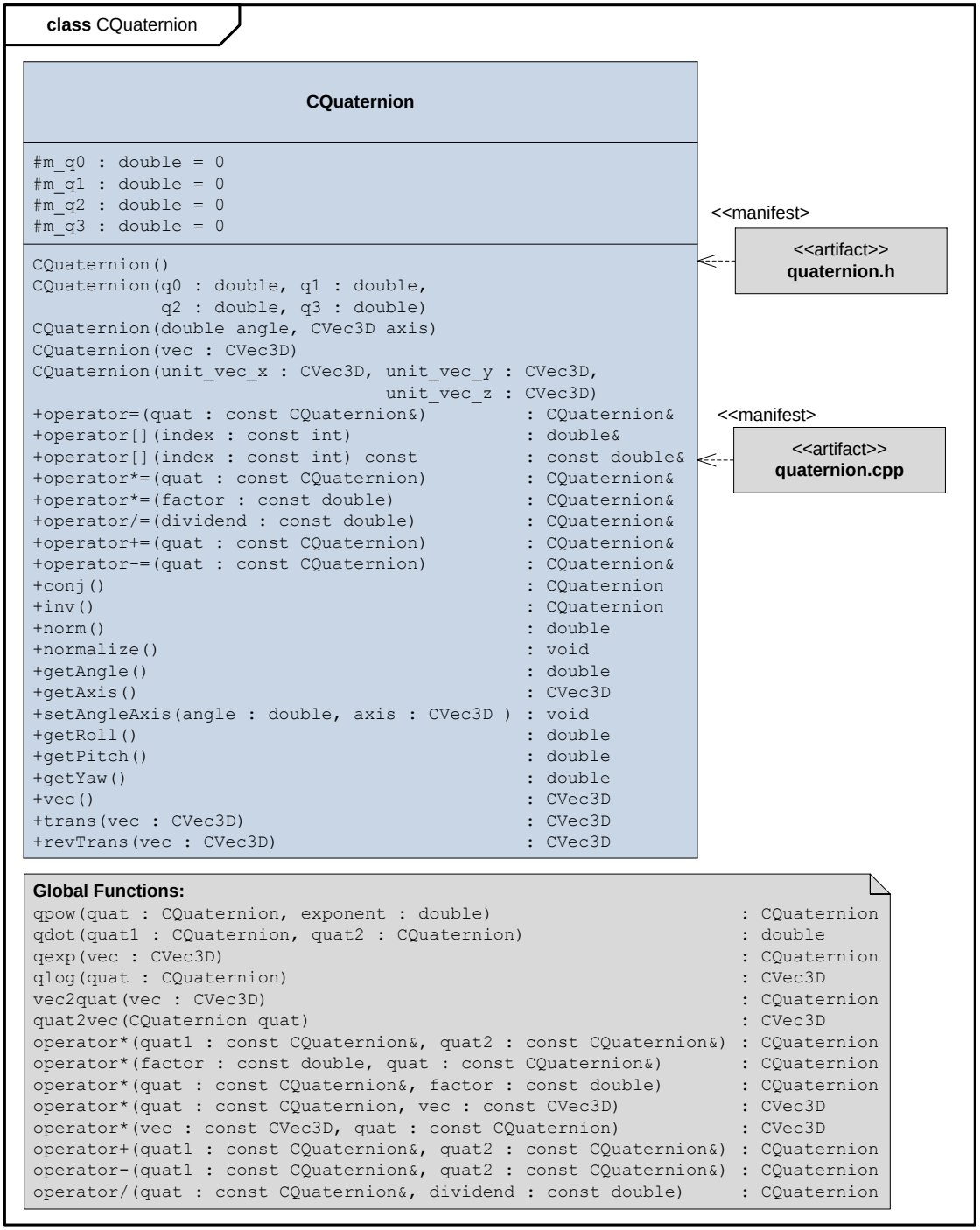

Figure C.24.: UML Class Diagram of class CQuaternion

ized by an instance of `CSpeedAccLimiter`. Save for the aforementioned differences, the meaning and usage of the `class`' methods is the same. Therefore, they are not described here. For more details, the reader may refer to `CCartTrajectoryGen` (e.g. `CAtomicTrajectoryGen`).

Context: An instance of `CQuatTrajectoryGen` is part of `CRobot` where it manages quaternion trajectory generation.

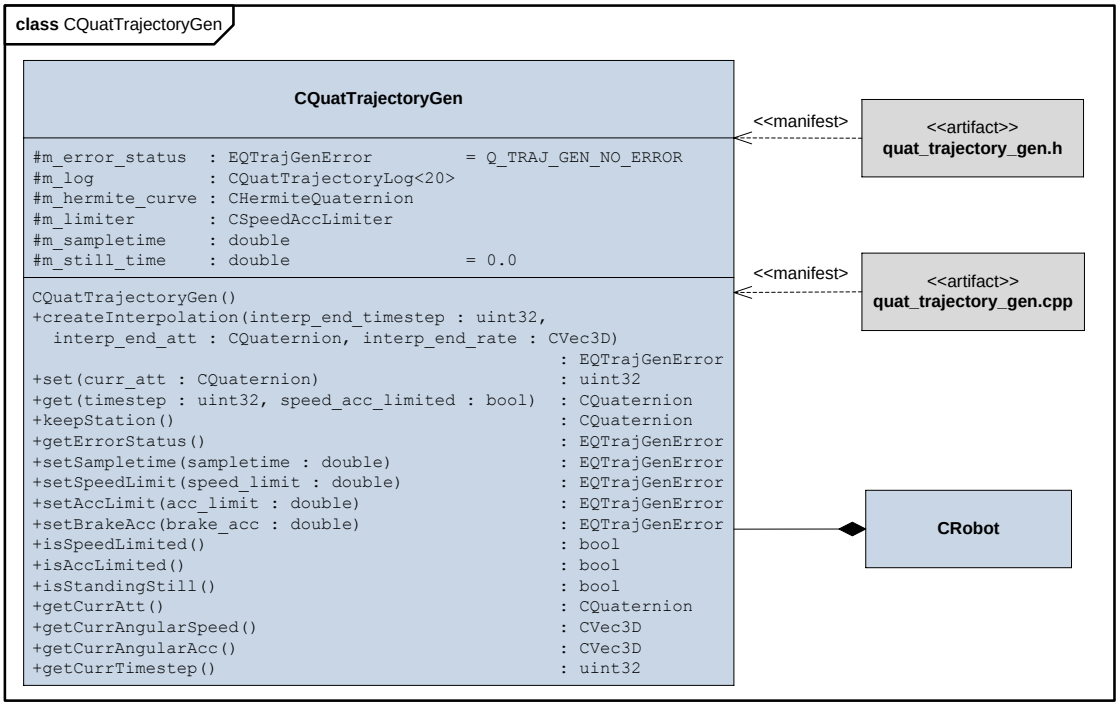

Figure C.25.: UML class diagram of class CQuatTrajectoryGen

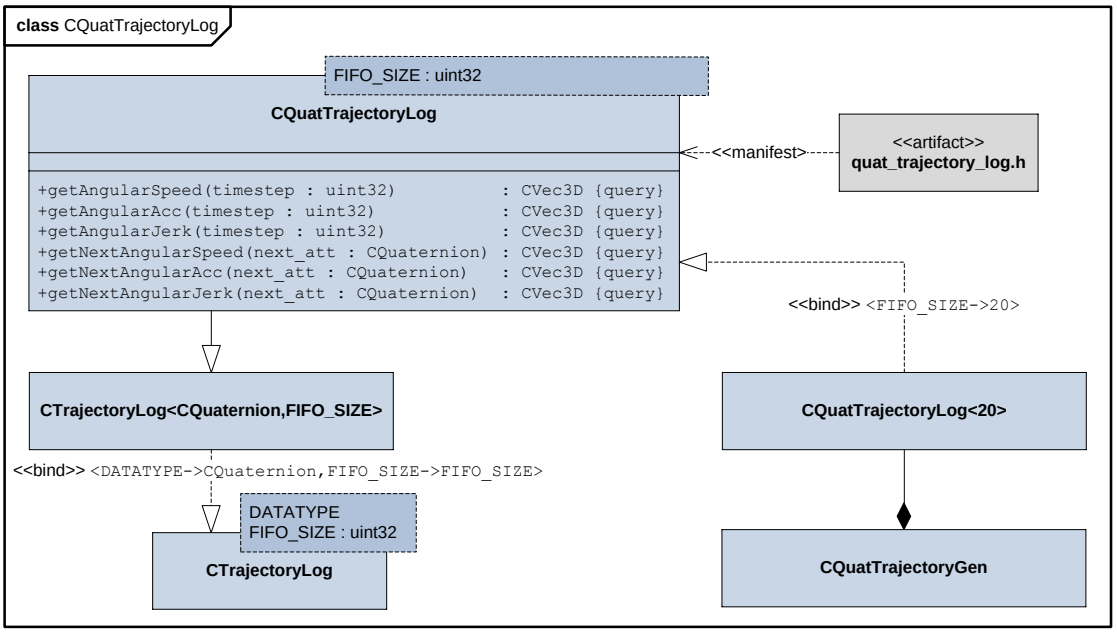

Figure C.26.: UML class diagram of class CQuatTrajectoryLog

C.24. class CQuatTrajectoryLog

Description: This `template` is derived from `CTrajectoryLog` with binding of `DATATYPE` to `CQuaternion`. Therefore it inherits the full functionality of `CTrajectoryLog` concering calculation of speed, acceleration and jerk, here for attitude in the form of quaternions. The purpose of `CQuatTrajectoryLog` is to extend these capabilites to the calculation of angular velocity, since this is the usual and practical way of describing the change of attitude over time. The quaternion interpola-

tion technique used requires angular velocities. As throuhgout this thesis, angular velocity is expressed in coordinates of the rotating frame, that is the spacecrafts' body-fixed frames. `CQuatTrajectoryLog` provides the following methods (which make use of the parent's methods):

`+getAngularSpeed(timestep : uint32) : CVec3D`
returns the angular velocity corresponding to the time step provided as argument. Valid values for `timestep` are as specified with `CTrajectoryLog::getSpeed(...)`. Consistent with (1.2), angular velocity is calculated as follows

$$\left(\omega^{Body}\right)_n = 2 \cdot (q)_n^{-1} \cdot (\dot{q})_n \tag{C.6}$$

`+getAngularAcc(timestep : uint32) : CVec3D`
returns the angular acceleration corresponding to the time step provided as argument. Valid values for `timestep` are as specified with `CTrajectoryLog::getAcc(...)`. As the derivative of (C.6), angular acceleration is calculated as follows

$$\left(\dot{\omega}^{Body}\right)_n = 2 \cdot \left[(\dot{q})_n^{-1} \cdot (\dot{q})_n + (q)_n^{-1} \cdot (\ddot{q})_n\right] \tag{C.7}$$

`+getAngularJerk(timestep : uint32) : CVec3D`
returns the angular jerk corresponding to the time step provided as argument. Valid values for `timestep` are as specified with `CTrajectoryLog::getJerk(...)`. As the derivative of (C.7), angular jerk is calculated as follows

$$\left(\ddot{\omega}^{Body}\right)_n = 2 \cdot \left[(\ddot{q})_n^{-1} \cdot (\dot{q})_n + 2 \cdot (\dot{q})_n^{-1} \cdot (\ddot{q})_n + (q)_n^{-1} \cdot (\dddot{q})_n\right] \tag{C.8}$$

And there are also the methods `getNextAngularSpeed(...)`, `getNextAngularAcc(...)` and `getNextAngularJerk(...)` which constitute the pendants of `getNextSpeed(...)`, `getNextAcc(...)` and `getNextJerk(...)` of `CTrajectoryLog`.

Context: This `class` is used as a member of `CQuatTrajectoryGen` which creates robot attitude trajectories by interpolation. Beside attitude this interpolation requires current angular velocity as starting condition. Therefore, `CQuatTrajectoryGen` is used to get current attitude, angular velocity and timestep.

C.25. class CRemoteSimClient

Description: `CRemoteSimClient` realizes the client part of RSP. All inputs specified in Tab.2.5 are implemented as according `set...` methods. All outputs specified in Tab.2.6 are implemented as according `get...` methods. All parameters specified in Tab.2.4 are provided to `CRemoteSimClient` via the initialization method `init(...)`. This method has to be called prior to class usage. In general, during simulation, the inputs have to be set first, then the function `runThisTimestep()` must be called and then the (updated) outputs can be read. The method `runThisTimestep()` is depicted in Fig.C.28. By calling `CClientStepTimer::beginThisTimestep()`, the client step timer is informed that a new time step has begun. Thereafter, the RSP client state machine - presented in Fig.2.13 in Sec.2.6.3.2 - is executed (protected method `runStateMachine`). Finally, all fields of `m_out_rs_packet` which are not set by the state machine individually are written to. The client state machine implemented by `runStateMachine` can be considered to be the core of `class CRemoteSimClient` and thus of the whole RemoteSim-Client Simulink block. There is a number of other protected methods, used by `runStateMachine`. `setRefTimesteps()` buffers the reference time steps (obtained from RSP client inputs) by calling `CClientStepTimer::setRefTimesteps(...)` of `m_client_step_timer` along with transmission of the `CMD_INIT` packet. Thus, the reference time steps provided by SCP at that point in time, are used for he whole simulation. `setInitCMD()`, `setTrajectoryCMD()` and `setForceTorqueCMD()` make the RSP packet to be transmitted a `CMD_INIT`, a `CMD_TRAJECTORY` or a `CMD_FORCE_TORQUE` packet respectively. The meaning of the member variables should be self-explanatory. There are variables to save parameters, inputs and outputs. Current client state is represented by `m_state`.

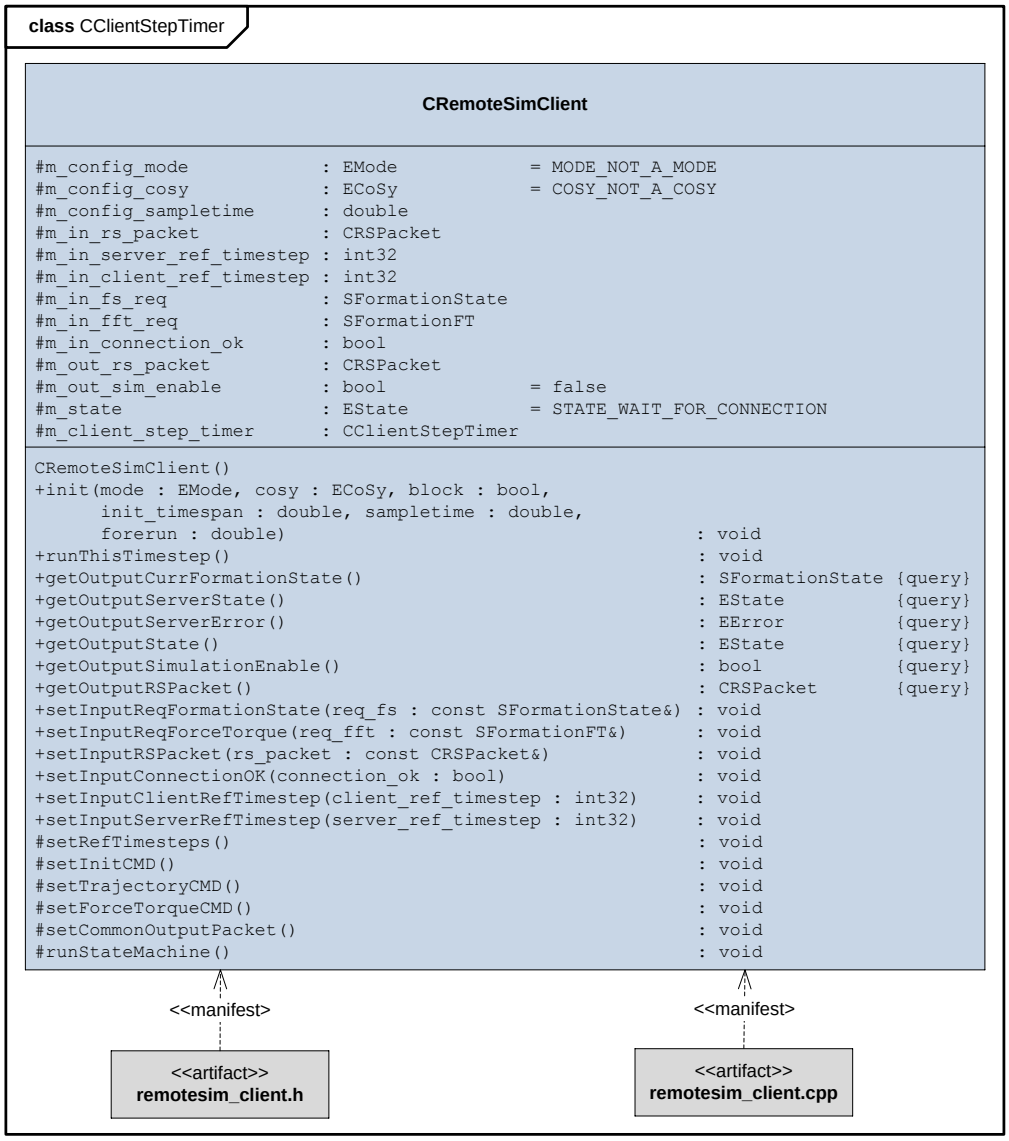

Figure C.27.: UML Class Diagram of class CRemoteSimClient

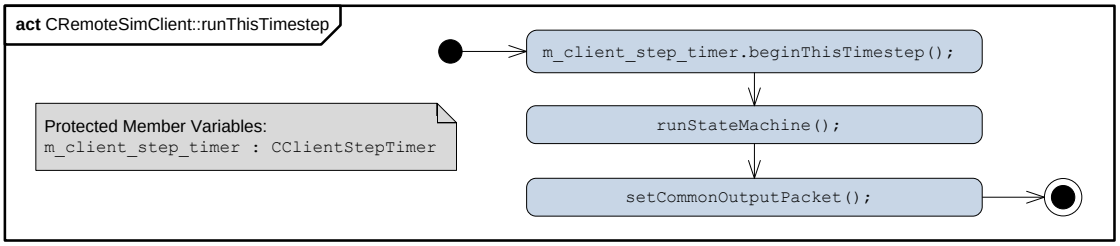

Figure C.28.: UML Activity Diagram of CRemoteSimClient method runThisTimestep()

Context: An instance of `CRemoteSimClient` is part of `SRemoteSimClientBlock`. It contains structures for spacecraft states and force/torque, a RSP packets representing the in-coming and out-going packets as well as an instance of `CClientStepTimer`. See Fig.C.3 for a summary.

C.26. class CRemoteSimServer

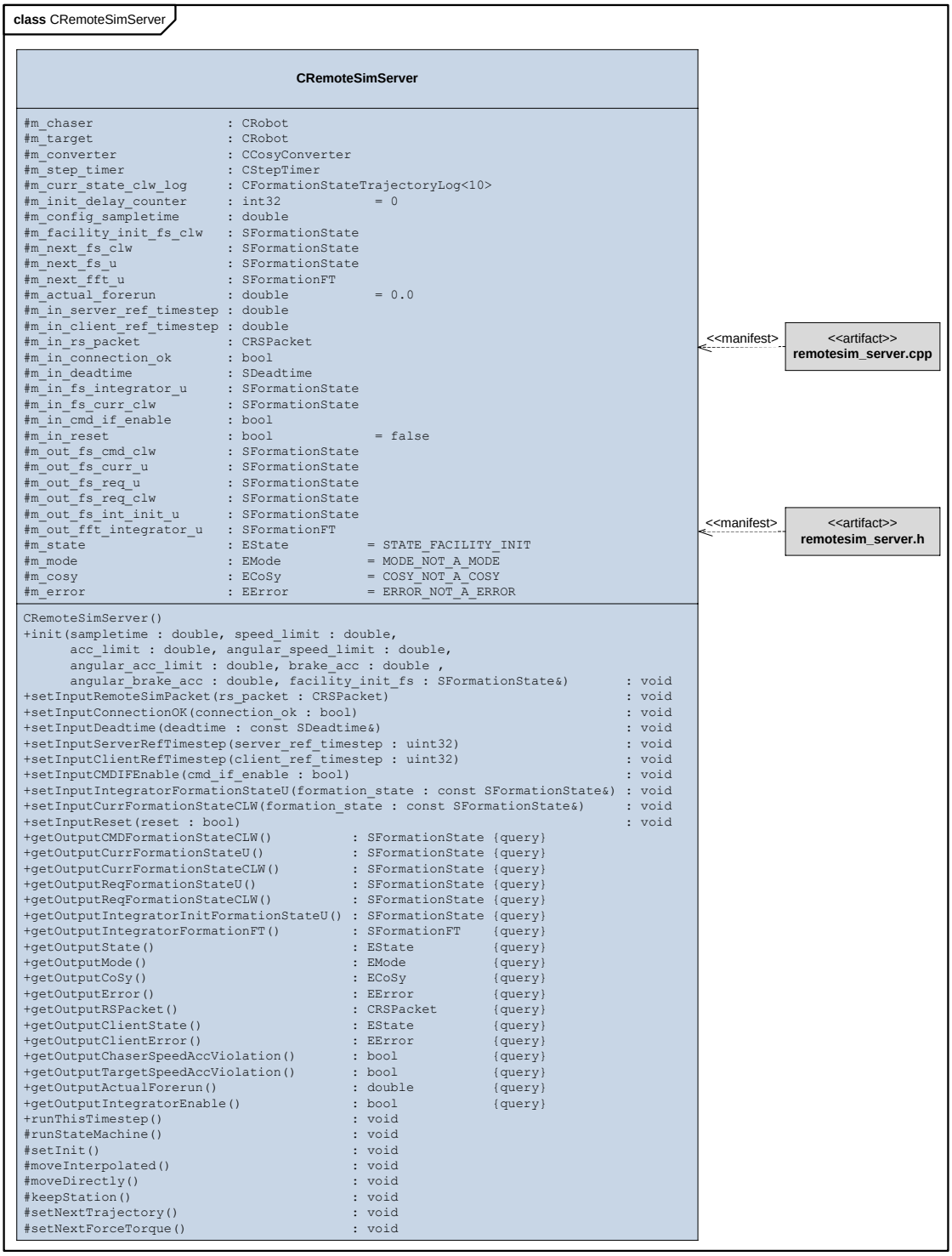

Figure C.29.: UML Class Diagram of class CRemoteSimServer

Description: The `class CRemoteSimServer` realizes the server part of SCP. It is the server counterpart to `CRemoteSimClient`. Similarly, all inputs specified in Tab.2.10 are implemented as according `set...` methods. All outputs specified in Tab.2.11 are implemented as according `get...` methods. All parameters specified in Tab.2.9 are provided to `CRemoteSimServer` via the initialization method `init(...)`, which method has to be called prior to class usage. At each time step, the inputs

have to be set first, then the function `runThisTimestep()` must be called and then the (updated) outputs can be read. Details of `runThisTimestep()` are discussed in Sec.3.4.3. The class diagram is given here. There are also several protected member methods. `runStateMachine()` executes

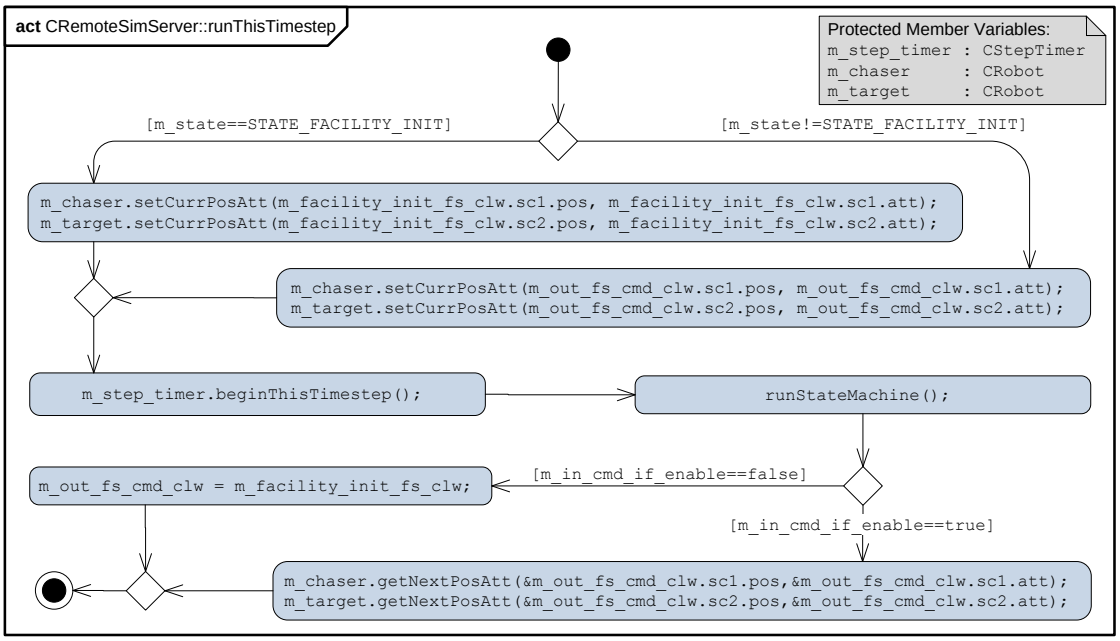

Figure C.30.: UML Activity Diagram of method `CRemoteSimServer::runThisTimestep()`

the RSP server state machine descriped in Fig.2.14. `setInit()` initializes `CStepTimer` for the coming simulation run and buffers initial states. `moveInterpolated()` calculates a new interpolated trajectory based on the next states buffered in the according member variables. `keepStation()` realizes the algorithm presented in Sec.2.6.4.3. `moveDirectly()` is used with `MODE_FORCE_TORQUE`. The spacecraft states set via `setIntegratorFormationStateU(...)` is put through to the robots directly. `setNextTrajectory(...)` and `setNextForceTorque(...)` buffer the values contained in a received CMD packet so that they can be used when next hit occurs.

Context: An instance of `CRemoteSimServer` is part of the `struct SRemoteSimServerBlock`. See Fig.C.1 for an overview.

C.27. class CRingQueue

Description: The `class CRingQueue` is similar to `CStaticFIFO`, except that it is allocated from dynamic memory, rather than from static memory. Therefore, the reader may refer to Sec.C.34 for a description of methods and member variables. `CRingQueue` is used as a fast buffer for the received continuous stream of bytes. The reason for having one version with dynamic memory is its use with SimCon. The length of the ring queue has to be determinable at run-time, since requirements dictate that SCP allows to choose the length of the vector to be transmitted (and received). The `class` is derived from `CSimConMemManagedClass` so that it can be used with the custom memory manager.

Context: An instance of `CRingQueue` is part of `CPacketBuffer`.

C.28. class CRobot

Description: The `class CRobot` represents a virtual robot in Software. Mainly, it combines an instance of `CQuatTrajectoryGen` and one of `CCartTrajectoryGen` thus realizing a complete translational and rotational trajectory for one robot. Therefore, many methods of `CRobot` resemble those of `CQuatTrajectoryGen` and `CCartTrajectoryGen`, except that position, speed, attitude and

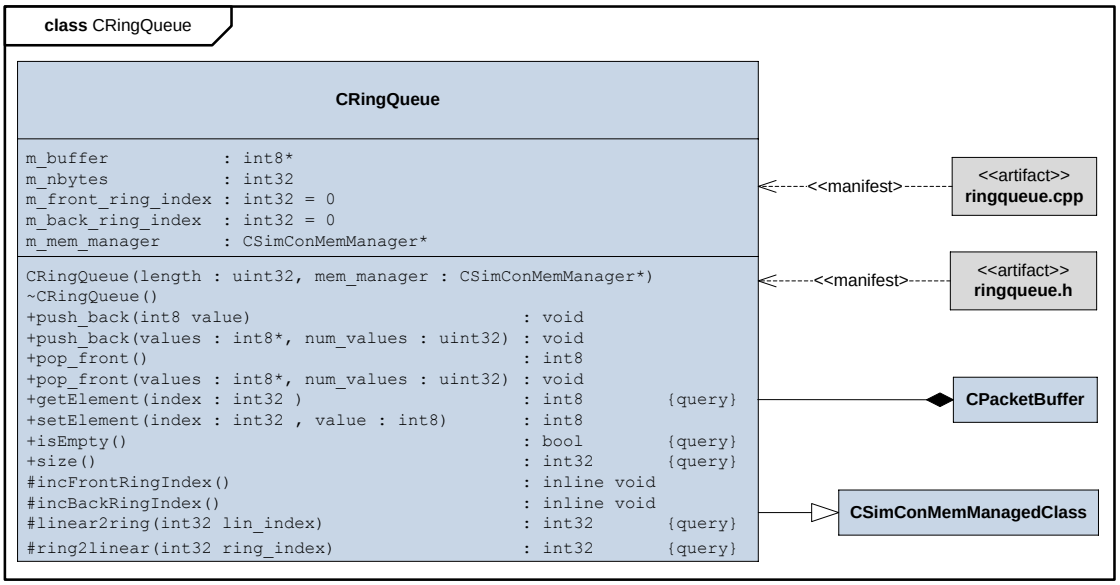

Figure C.31.: UML Class Diagram of class CRingQueue

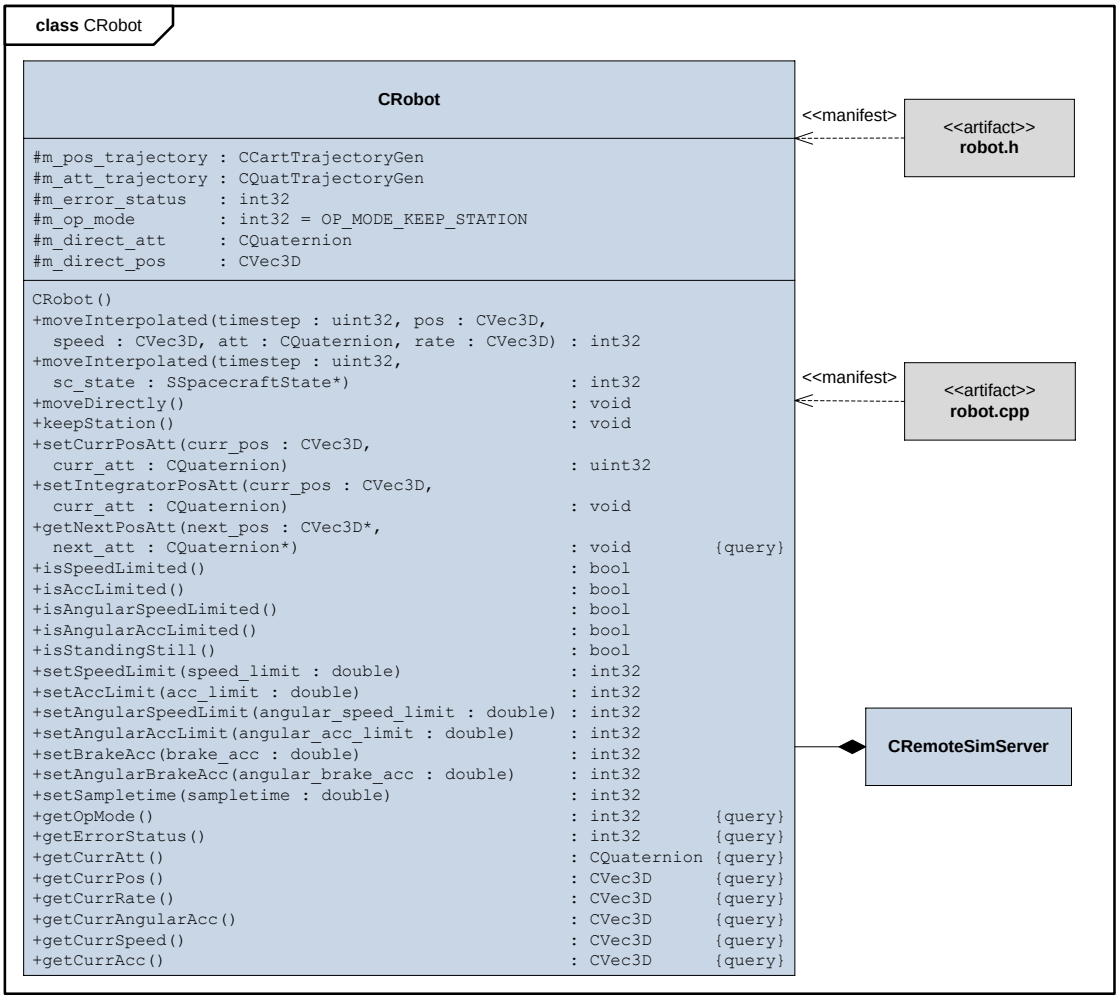

Figure C.32.: UML class diagram of class CRobot

angular velocity are handled together as SFormationState at some points. In fact most of the methods equal precisely its counterparts of CQuatTrajectoryGen and CCartTrajectoryGen and are therefore not described in any more detail here. However, beside moveInterpolated(...) and keepStation(...), there is another way of operating the robots: moveDirectly(). From the point this method is called, CRobot simply puts through the trajectory which is set by the method setIntegratorPosAtt(...) (at each time step), coming from the integrator. This is used with MODE_FORCE_TORQUE. The member variables m_direct_pos and m_direct_att are used to buffer this trajectory.

Context: Two instances of CRobot are part of CRemoteSimClient, one representing the chaser spacecraft and one representing the target spacecraft.

C.29. class CRSPacket

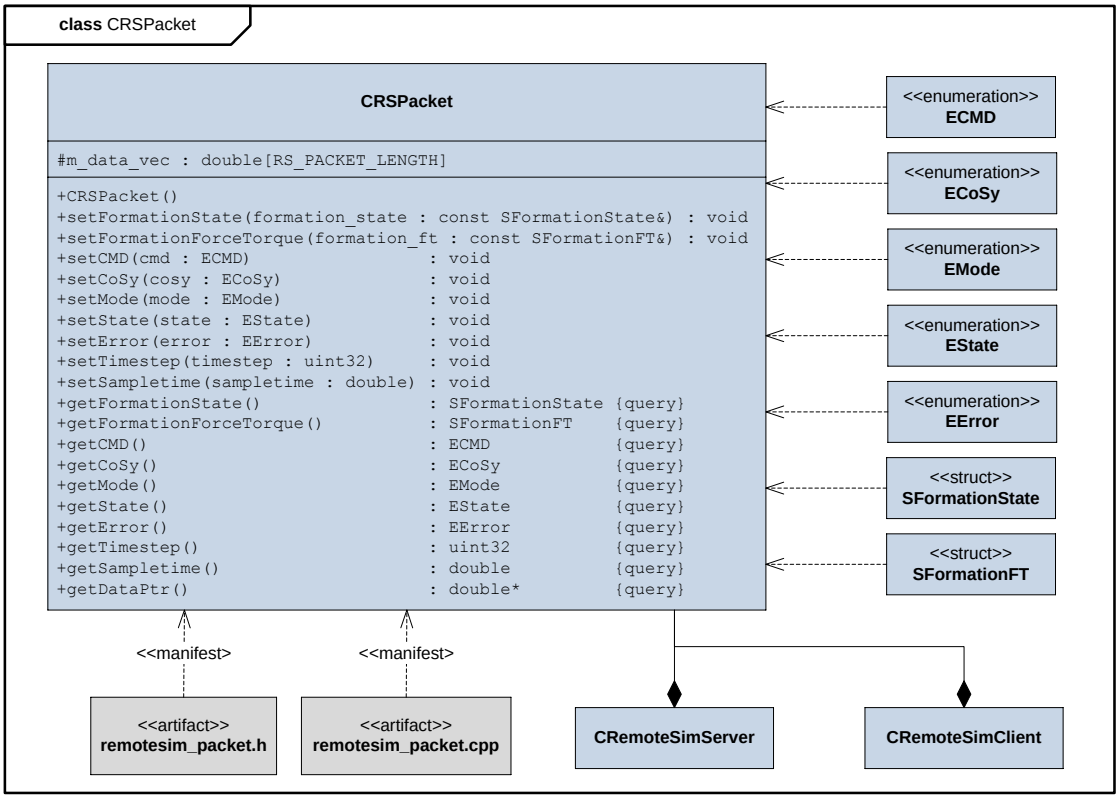

Figure C.33.: UML Class Diagram of class CRSPacket

Description: CRSPacket implements the RSP packet, defined in Sec.2.6.1. The fact that a RSP packet is essentially a 1D array of values is reflected by the only member variable m_data_vec, an array of type double. There are only two types of member methods: set methods are for writing values to the different fields of the packet and get methods are for reading these values from the packet. The RSP packet's structure is specified by a number of #define constants listed in remotesim_packet_spec.h.

Context: class CRSPacket is used by CRemoteSimClient and CRemoteSimServer.

C.30. class CSimCon

Description: The class CSimCon implements the Simulation Connection Protocol. An outline of the responsibilites of its member variables has already been given in Sec.3.5. All inputs specified in Tab.2.2 are implemented as according set... methods. All outputs specified in Tab.2.3

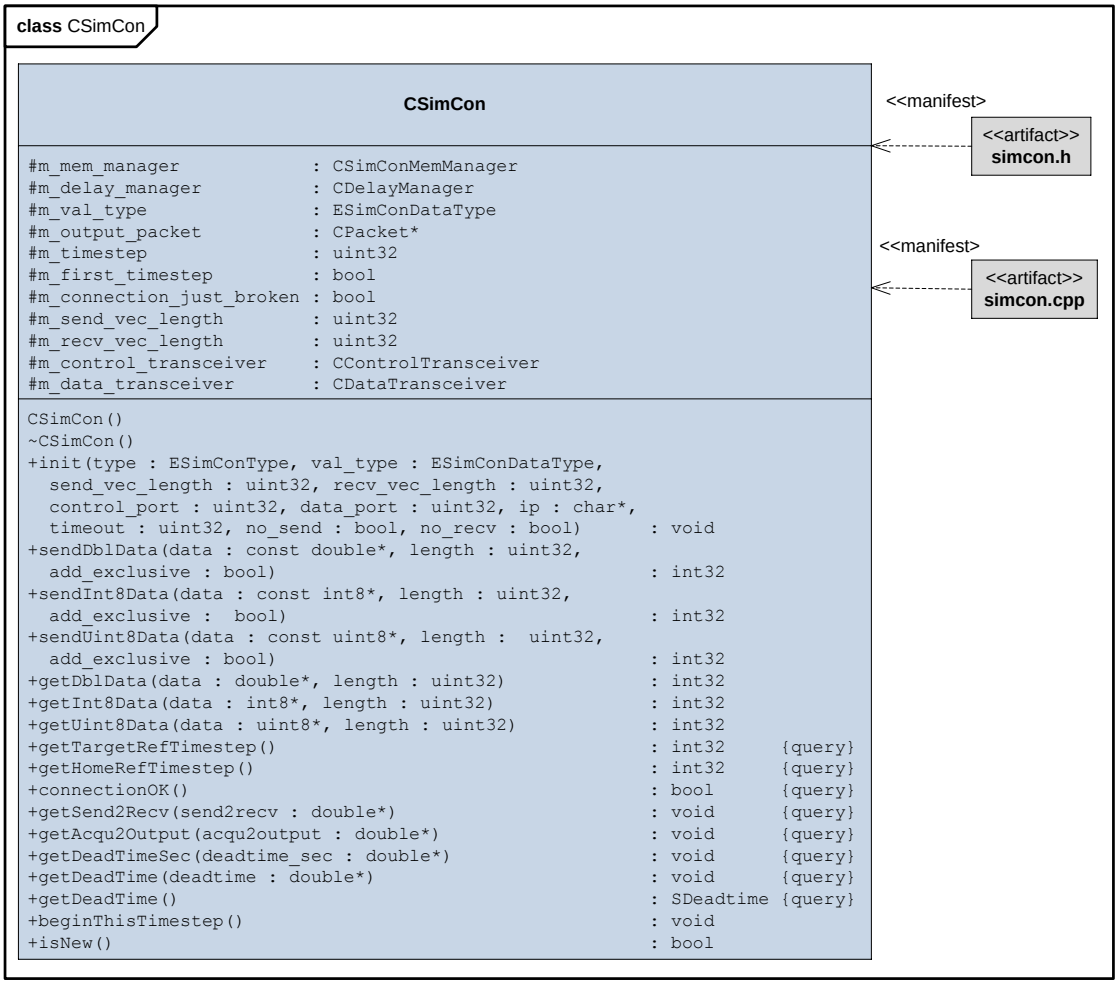

Figure C.34.: UML Class Diagram of class CSimCon

are implemented as according `get...` methods. All parameters specified in Tab.2.1 are provided to `CSimCon` via the initialization method `init(...)`. This method has to be called prior to class usage. `isNew()` indicates that there is received data available that has not been read yet. `beginThisTimestep()` is to be called at the beginning of each time step. The method is described in Fig.C.35. First, the current time step is logged. The method is to be executed at the beginning of each time step. Therefore, `m_timestep` is not increased at the first time step, since at that point 0 is the correct value. Next, it is checked whether the connection is intact. For that purpose, `CControlTransceiver` provides the method `connectionOK()`. If the connection is indeed intact, then an according flag is set (`m_connection_broken = false`). If the connection is broken, `m_control_transceiver` and `m_data_transceiver` are commanded to try to recreate the connection via method `reconnect()`. Finally, `m_control_transceiver` is informed that a new time step has begun by calling `CControlTransceiver::beginThisTimestep()`.

Context: An instance of `CSimCon` is part of `SRemoteSimServerBlock` and of `SRemoteSimClientBlock`.

C.31. class CSimConMemManagedClass

Description: `CSimConMemManagedClass` is an abstract `class`. All `classes` that are to be used with the custom memory manager `CSimConMemManager` have to be derived from this `class`. In essence, it overloads the placement `new` operator and introduces the method `destroy(...)` as the

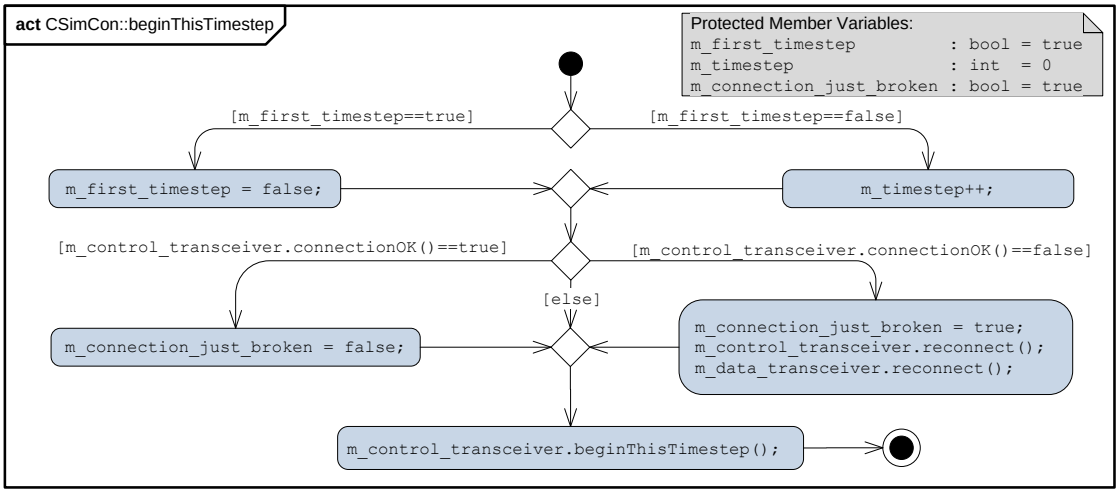

Figure C.35.: UML Activity Diagram of CSimCon method beginThisTimestep()

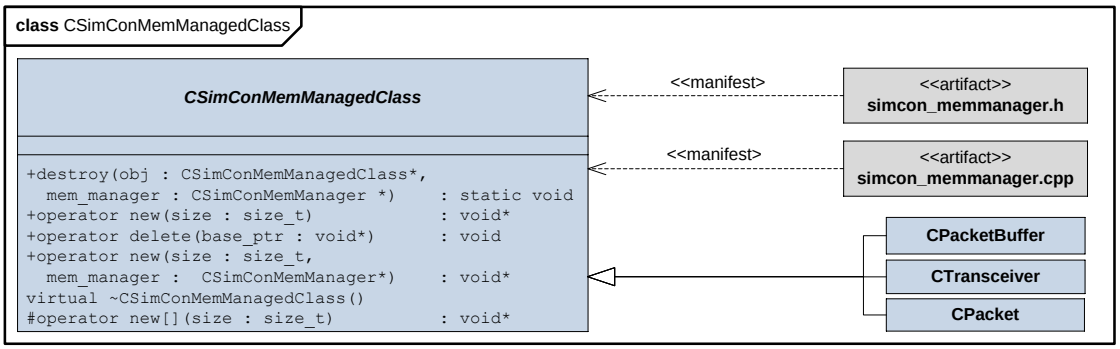

Figure C.36.: UML Class Diagram of class CSimConMemManagedClass

pendant to `delete`. The argument to placement `new` and the second argument to `destroy(...)` has to be a pointer to the memory manager.

Context: This is an abstract `class` and is not instanciated. `CPacket`, `CRingQueue` and `CPacketBuffer` are derived from this class.

C.32. class CSimConMemManager

Description: `CSimConMemManager` realizes a custom manager for dynamic memory allocation. All `classes` that should be used with the memory manager must be derived from `CSimConMemManagedClass`. Background: VxWorks' dynamic memory management leads to fragmentation if frequent memory allocation/deallocation. The memory manager avoids this problem by managing a block of memory itself and by implementing a defragmentation algorithm. There are only a few public methods. `allocMemory(...)` has to be called upon instanciation to allocate the memory the manager shall work with. Note that this memory block is allocated from the system dynamically. So there is dynamic memory allocation, but only once at the beginning so that VxWorks can deal with it, without problem. Any subsequent call to `allocMemory(...)` is ignored. The method `isInitialized()` returns `true` if `allocMemory(...)` has already been called. Then `allocMemBlock(...)` returns a `void` pointer to a block of memory of the specified size or `NULL` if none is available. `freeMemBlock(...)` makes the block available again, whereby the address of the memory block must be given as argument. `CSimConMemManager` uses a list of instances of `CMemoryBlock` to manage the memory. The order of the elements in the list is similar to the sequence of blocks in the managed memory. Each block is either free or used. At the beginning, only one free block is present. `allocMemBlock(...)` uses `split(...)` to split the block in a used one with the

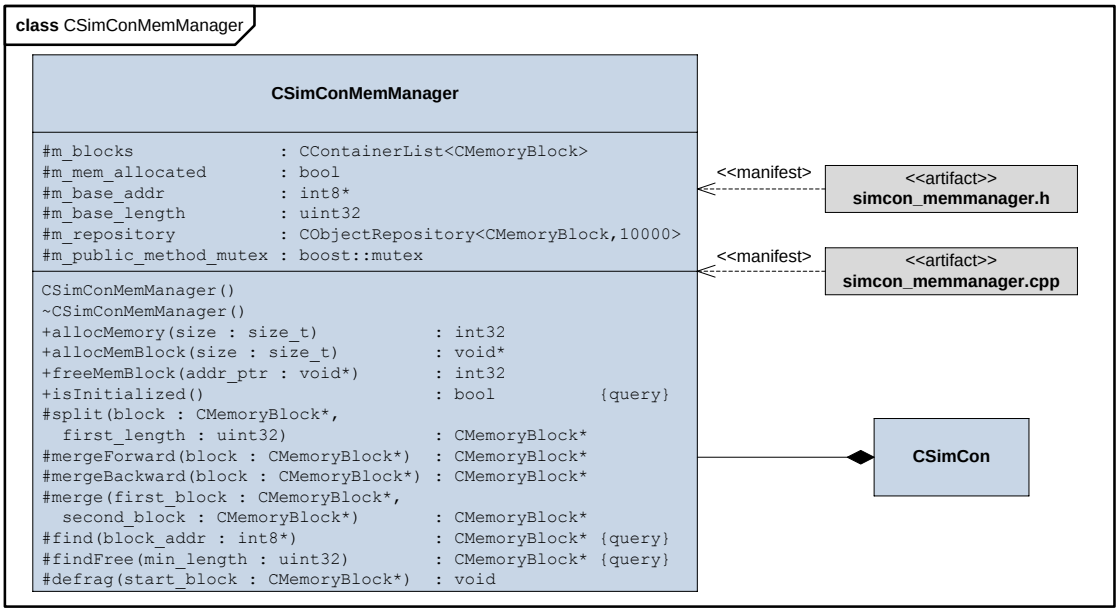

Figure C.37.: UML Class Diagram of class CSimConMemManager

desired size and a free one. The next call to `allocMemBlock(...)` uses `findFree(...)` and again `split(...)` to create two blocks from the found free one. At some point, there is a list of memory blocks, one ending where the next begins, some free and some in use. When `freeMemBlock(...)` is called with the address of the block in memory, the instance of `CMemoryBlock` is accessed directly. (This kind of fast access is possible due to the use of `CContainerList`.) The block is marked as free and it is checked, whether the previous and/or next one is also free. In that case `mergeBackward(...)` and/or `mergeForward(...)` combine the free blocks to form one larger free block. In this way, it is ensured that contiguous blocks of memory are always as large as possible, thus minimize fragmentation.

Context: An instance of `CSimConMemManager` is part of `CSimCon` and is used for allocating memory for `CPacket`, `CRingQueue` and `CPacketBuffer`.

C.33. class CSpeedAccLimiter

Description: `CSpeedAccLimiter` serves two pruposes: Determine trajectory values with limited speed and acceleration based on values that have not been limited yet and calculation of trajectory values to realize keep station, e.g. to brake the robots to zero speed. Both tasks are dealt with for translational and rotational movement (quaternion). First, required parameters have to be provided via `setSpeedLimit(...)`, `setAccLimit(...)`, `setSampletime(...)` and `setBrakeAcc(...)`. Then, `getLimitedVal(...)` returns the speed/acceleration-limited translational position based on current value, next value (without limitation), current speed and next speed (without limitation). Similarly, `getLimitedQuat(...)` returns the speed/acceleration limited rotational attitude based on current attitude, next attitude (without limitation), current angular velocity and next angular velocity (without limitation). After calling one of these methods, `isSpeedLimited()` and `isAccLimited()` signal whether speed and/or acceleration have been modified to comply with the specified limits. The method `getStationKeepingVal(...)` uses current position and speed to calculate the next position thereby reducing robot speed C^1 continuously according to (2.22) and (2.23). The method `getStationKeepingQuat(...)` uses current attitude and angular velocity to calculate the next attitude thereby reducing robot angular velocity C^1 continuously according to (2.24), (2.25), (2.26) and (2.27).

Context: An instance of `CSpeedAccLimiter` is part of `class CAtomicTrajectoryGen` and of `class CQuatTrajectoryGen`.

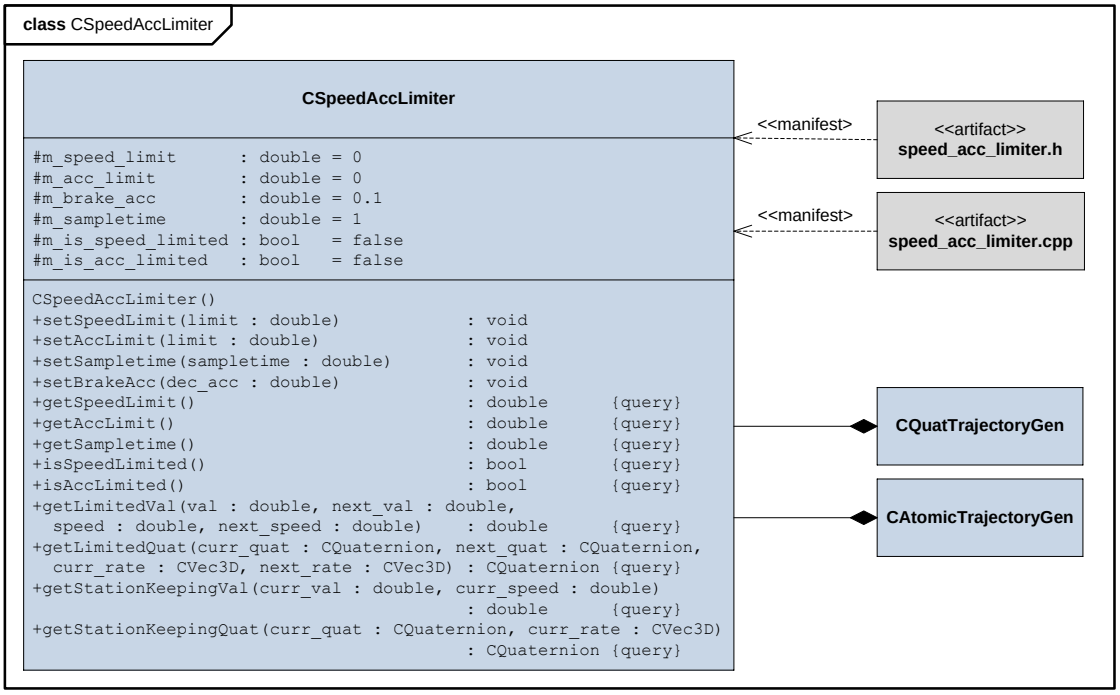

Figure C.38.: UML Class Diagram of class CSpeedAccLimiter

C.34. class CStaticFIFO

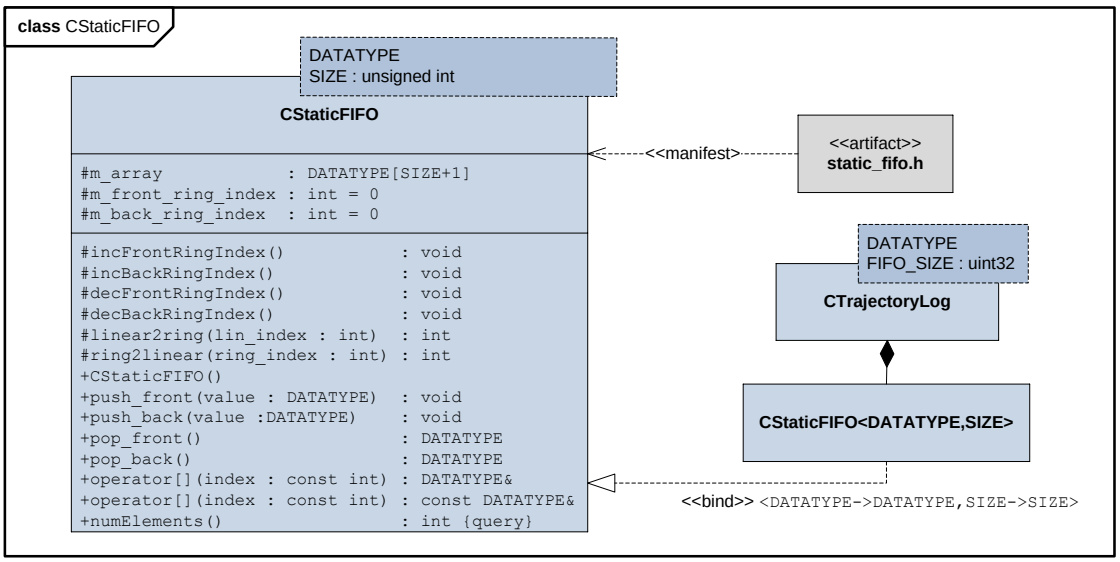

Figure C.39.: UML Class Diagram of class CStaticFIFO

Description: The template `class CStaticFIFO` realizes a First-In-First-Out buffer thereby avoiding any dynamic memory allocation. Static referes to the fact that any data is put on the stack rather than on the heap. The FIFO functionality is the class' primary purpose. However, there are methods to put and pop elements from both ends of the queue. Thus, a LIFO (Last-In-First-Out) can also be realized. The first template argument is the datatype (DATATYPE) of the objects and the second specifies the number of objects (SIZE) to be stored in the FIFO. The FIFO mechanism is based on the ringbuffer concept. The ringbuffer is represented by a simple contiguous array in memory (`m_array`).

Member variables mark the first (m_front_ring_index) and last element (m_back_ring_index) in the ringbuffer. Therefore, the minimum number of elements is zero while the maximum number of elements is limited by the template argument SIZE. Since there is no dynamic memory allocation or deallocation necessary for pushing or popping objects and merely start and end marking on the ringbuffer are changed CStaticFIFO is very fast.

A number of protected methods form the basis: incFrontRingIndex() increases m_front_ring_index by one. If the index should then exceed the dimensions of m_array it is set 0, thus moving "in the ring". Respectively, decFrontRingIndex() decreases m_front_ring_index by one. If the index should then be negative (after having been decreased from 0 to -1) it is set SIZE-1. The protected methods incBackRingIndex() and decBackRingIndex() behave similarly but alter m_back_ring_index. linear2ring(...) takes an index refering to an element in the FIFO that is an index between 0 and SIZE and returns the index pointing to the associated element of m_array. Example: The first element in the ringbuffer (lin_index) may be located at position 5 (return value) in memory. ring2linear() is the counterpart of linear2ring. It takes an index refering to an element in m_array and returns the index of the asscociated element in the ringbuffer.

The public methods push_front(...) and push_back(...) allow to add one element to the FIFO at the front or at the back. Thereafter this element constitutes the first element, e.g. the last element in the fifo. If the FIFO was already full, the former last element, e.g. first element is deleted from the FIFO. If the FIFO was not full, its number of elements is increased by one. The public methods +pop_front() : DATATYPE and +pop_back() : DATATYPE return the first, e.g. the last element in the FIFO. It also removes this element so that the number of elements is reduced by one. With an empty FIFO this method returns an invalid element of unspecified state. +operator[](...) allows read-write acces to the element addressed by the argument index. If index is invalid, the returned element is in unspecified state. Finally, numElements() returns the number of elements currently stored in the FIFO.

Context: An instance of the template is used in template CTrajectoryLog, where it stores the latest SIZE trajectory values, position (cartesian) and attitude (quaternion) with the according binds.

C.35. class CStaticList

Description: The template CStaticList realizes a list with the usual functionalities. In contrast to the templates of the C++ standard library, CStaticList doesn't use dynamic memory allocation to avoid problems with VxWorks. Rather, a maximum number of elements is given as template argument MAX_SIZE. The length of the list depends on the number of elements inserted, but the amount of memory the list requires always equals the maximum size of the list. The other template argument specifies the DATATYPE of the elements to be stored. To realize the list, CStaticList uses instances of CListElement. These instances are not allocated dynamically (which would counteract the fact that the list should be located in static memory) but uses an instance of CObjectRepository as a provider for list elements. The method pop_front() returns and removes the first element in the list. The method push_back(...) inserts an element at the end of the list. remove(...) removes the element with the index given as argument. insert(...) puts an element into the list where the index given as argument determines the elements position in the list. The element which had this index before, is shifted backwards in the list. get(...) returns the element with the specified index without removing it. size(...) returns the number of elements in the list, not the maximum size. isEmpty() is true, if the list is empty. Other protected methods support the public access methods and avoid redundancy.

Context: An instance of CStaticList is part of CPacketFIFO.

C.36. class CStepCounter

Description: CStepCounter is a small extension of a positive integer variable used as a counter. The operator++ can be used to increas its value. By default, the counter starts at 0 with a counting interval of 1 and the first call to operator++ has no effect. Thus, time steps can be comfortably counted at the beginning of a routine executed at each simulation step, the time step being 0 during

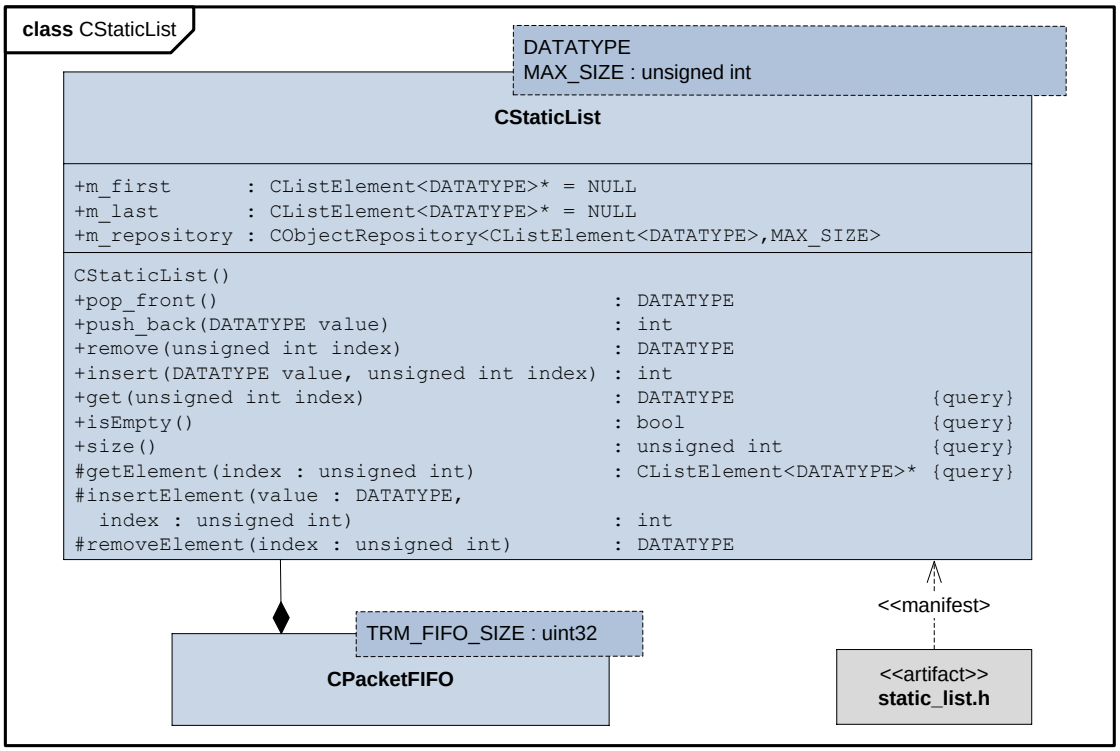

Figure C.40.: UML Class Diagram of class CStaticList

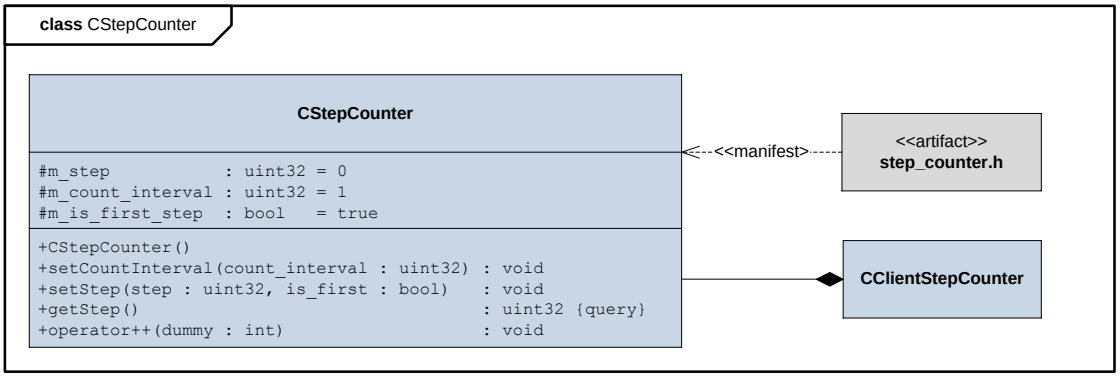

Figure C.41.: UML Class Diagram of class CStepCounter

the first execution. With `setCountInterval(...)` the counting interval can be changed to any positive integer. Moreover, the method `setStep(...)` allows for setting a specific value as the current counter value and for determining if this shall be considered the first step. This way, the mechanism of omitting the first `operator++` can be switched off. The counter value can be obtained by calling `getStep()`. The meaning of the three member variables is self-explanatory.

Context: An instance of `CStepCounter` is part of `class CClientStepTimer`.

C.37. class CStepTimer

Description: The `class CStepTimer` handles all timing tasks RemoteSim-Server relies on. At first, the maximum CMD delay is to be set by executing `setMaxCMDDelay(...)`. And before using the `class` during a simulation step, the method `beginThisTimestep()` must be called (at each time step) which does nothing but count time steps. At any point, `CStepTimer` can be initial-

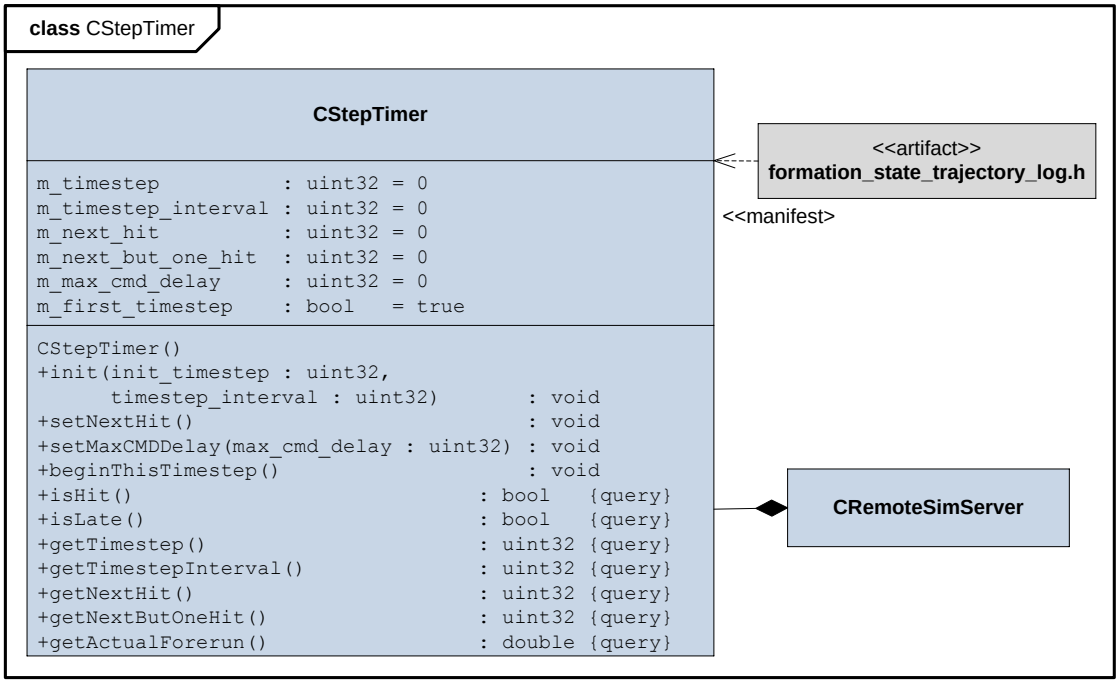

Figure C.42.: UML Class Diagram of class CStepTimer

ized by method `init(...)` with an initial time step (which equals $n^S_{init,s}$, see Sec.2.6.3.3) and a time step interval (which equals ρ_{sample}, see Sec.2.6.2) arguments. "Pushing hits", as descriped in Sec.2.6.2, is carried out by `setNextHit()`. `isHit()` indicates when next hit has been reached. Its return value is true as long as current time step is at least as large as next hit. `isLate()` indicates that number of steps passed since next hit was reached is greater then the maximum CMD delay. (Then it is assumed that something is wrong with the remote simulation or the connection.) `getNextHit()`, `getNextButOneHit()` and `getTimestep()` return the associated values. The method `getActualForerun()` returns the actual forerun. It is the number of steps between next hit and the time step when a (punctual) CMD packet is received, with respect to the client step ratio. The `class` assumes that the receipt event cooincides with the call to `getActualForerun()`, hence the relevant time step is the current time step.

$$\nu_{actual} = \frac{n^S_{hit,s} - n^S_{curr}}{\rho_{sample}} \tag{C.9}$$

Context: An instance of `CStepTimer` is part of `class CRemoteSimServer`.

C.38. class CTSVar

Description: This simple `template` realizes a thread safe primitive variable. A mutex of type `boost::mutex` is locked while accessing the variable via `set(...)` and `get()` methods. Background: In the multi-threaded `class CSimCon` many primitive variables are accessed at the same time. It cannot be garanteed that access of a primitive variable is not interrrupted by a change of the thread.

Context: Instances of this `template` are used with most of the active `classes` that are part of `CSimCon`.

C.39. class CTrajectory

Description: `CTrajectory` represents a 1D translational trajectory, determined by C^1 continuous interpolation, as described in Sec.2.6.4.5. The `class CLinEqusSys`, in the form of member variable

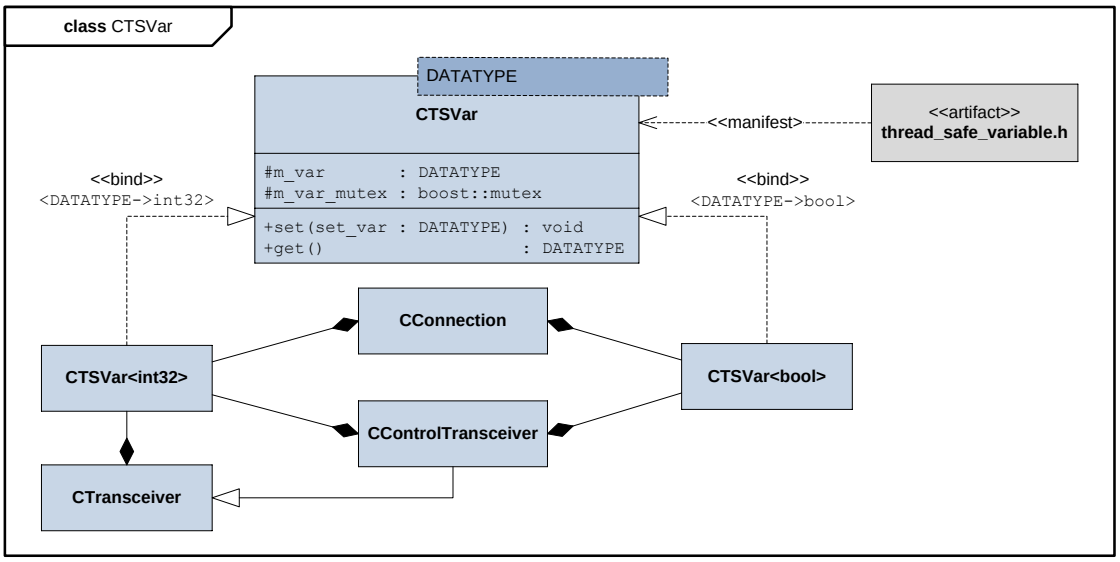

Figure C.43.: UML Class Diagram of class CTSVar

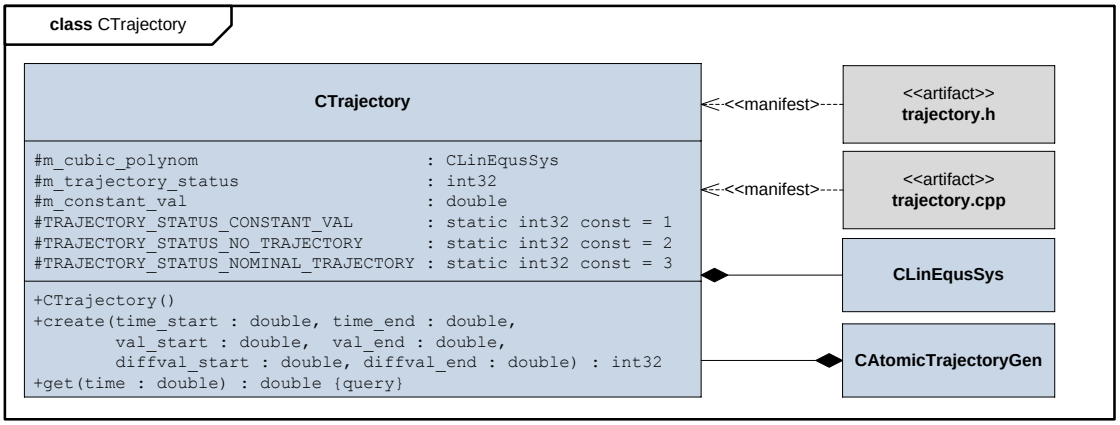

Figure C.44.: UML Class Diagram of class CTrajectory

`m_cubic_polynom`, serves as a solver for the linear equation system associated with interpolation. Usage of the `class` is very simple. Method `create(...)` determines a trajectory by interpolation with start time, position and speed as well as end time, position and speed as arguments. The method `get(...)` can then be used to obtain the position at a specific point of time (which must be provided as argument). The majority of work is carried out by the member variable `m_cubic_polynom`.

Context: An instance of `CTrajectory` is part of the `class CAtomicTrajectoryGen` and is used for generating a 1D translational trajectory between next hit and next-but-one hit.

C.40. class CTrajectoryLog

Despription: The `template CTrajectoryLog` can be used to log a specified number of the latest elements, representing a trajectory, set via `addCurrValue(...)` and calculate speed, acceleration and jerk from these elements. The method `addCurrValue(...)` assumes that each call corresponds to one simulation time step. Thus, time steps are counted internally. Before using the class, the simulation's sample time must be set by calling `setSampletime(...)`. The first `template` argument is the type of the elements which constitute the trajectory. In general, the only restriction to this type is that it has to support the binary operators $+, -, *$ and $/$ as well as multiplication with a scalar. The second `template` argument specifies the number of latest elements to be stored. Logging of

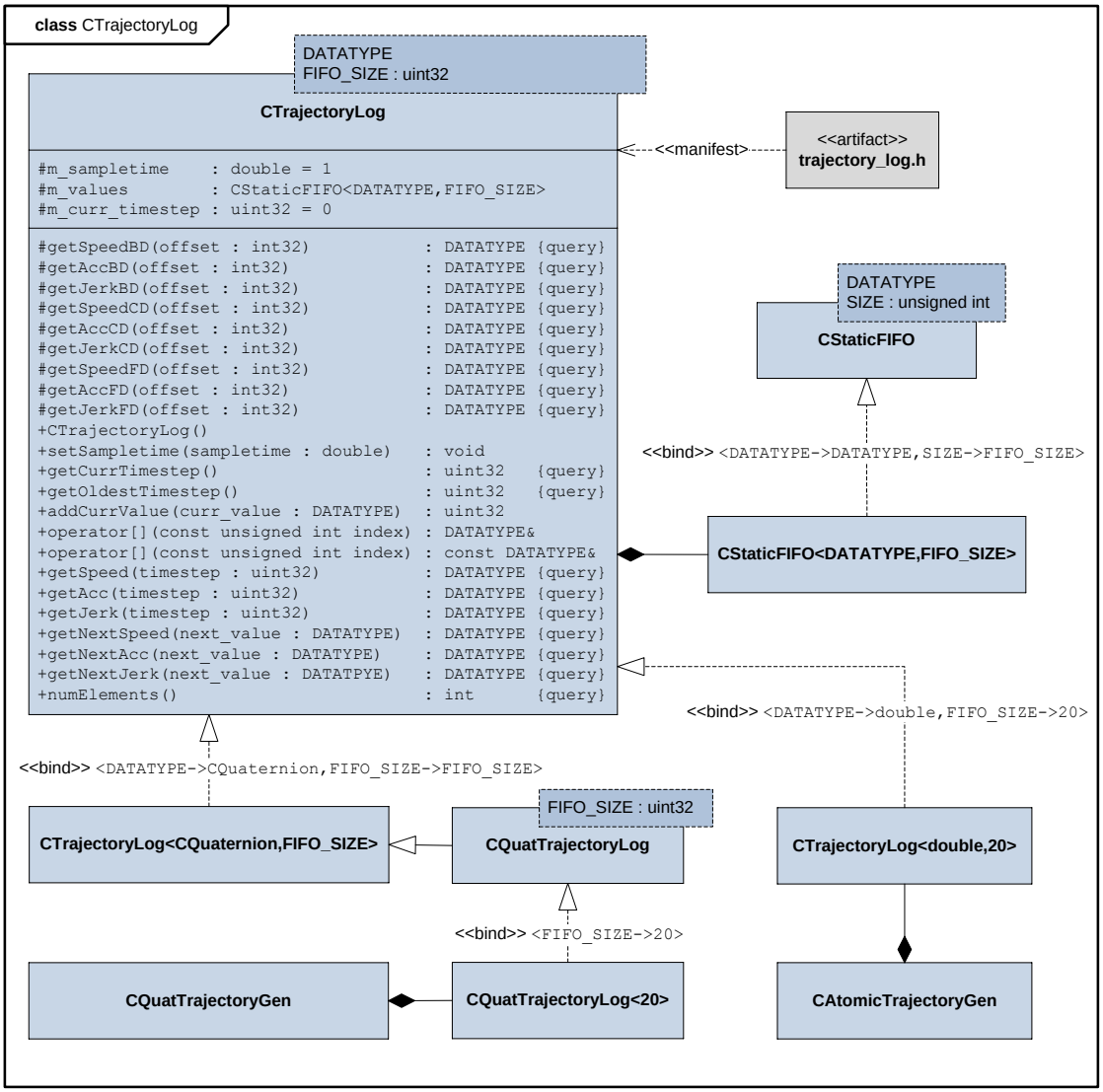

Figure C.45.: UML Class Diagram of class CTrajectoryLog

the elements is achieved with a FIFO (`#m_values : CStaticFIFO<DATATYPE,FIFO_SIZE>`). The meaning of the member variables `m_sampletime` and `m_current_timestep` is self-explanatory. Not only can current speed, acceleration and jerk associated with the most recently added element be calculated but also speed, acceleration and jerk corresponding to any of the elements stored in the FIFO. Thus, since each element is associated with a certain timestep, past speed, acceleration and jerk can be determined. Depending on the position in the FIFO, these derivatives are calculated using Backward Difference (BD), Central Difference (CD) and Forward Difference (FD) with BD prefered. Example: In case of the current speed BD is used (no future values known). If corresponding to the last element in the FIFO, FD is used (no past values known). And finally, if somewhere in the middle fo the FIFO, CD is used (future and past values known). Acceleration and jerk may need values not only one step ahead and/or back but even two. All equations, that is the finie difference coefficients, are taken from [12]. The methods which return speed, acceleration and jerk take the absolute time step as an argument and not a relative index. The class assumes that every call to `addCurrValue(...)` marks a new time step with the first call at timestep 0. Thus the class counts time steps internally.

Here, the (protected) methods for calculating speed, acceleration and jerk are presented one by one in order to give the individual formular used. The symbols used: n_{offset} denotes a timestep offset,

0 refering to the current element in the FIFO, -1 the one before the current and so forth. (Note: Internally, `CStaticFIFO` takes positive indices to address the elements stored. But at one level of abstraction higher it makes more sense to use negative indices for past elements.) $k(n_{offset})$ is the value of the element associated with n_{offset} in a most general sense. It may be scalar position, vector position or even attitude, etc..Δt_{sample} is the sampletime. l is the number of elements currently stored in the FIFO. The formulas are taken from [12].

`#getSpeedBD(offset : int32) : DATATYPE`
returns speed at the time step corresponding to `offset` calculated as Backward Difference. $n_{offset} \in [-(l-2), 0]$ must hold. If the offset is invalid, the state of the returned element will be undefined.

$$\dot{k}(n_{offset}) = \frac{k(n_{offset}) - k(n_{offset}-1)}{\Delta t_{sample}} \tag{C.10}$$

`#getAccBD(offset : int32) : DATATYPE`
returns acceleration at the time step corresponding to `offset` calculated as Backward Difference. $n_{offset} \in [-(l-3), 0]$ must hold. If the offset is invalid, the state of the returned element will be undefined.

$$\ddot{k}(n_{offset}) = \frac{k(n_{offset}) - 2 \cdot k(n_{offset}-1) + k(n_{offset}-2)}{(\Delta t_{sample})^2} \tag{C.11}$$

`#getJerkBD(offset : int32) : DATATYPE`
returns jerk at the time step corresponding to `offset` calculated as Backward Difference. $n_{offset} \in [-(l-4), 0]$ must hold. If the offset is invalid, the state of the returned element will be undefined.

$$\dddot{k}(n_{offset}) = \frac{k(n_{offset}) - 3 \cdot k(n_{offset}-1) + 3 \cdot k(n_{offset}-2) - k(n_{offset}-3)}{(\Delta t_{sample})^3} \tag{C.12}$$

`#getSpeedCD(offset : int32) : DATATYPE`
returns speed at the time step corresponding to `offset` calculated as Central Difference. $n_{offset} \in [-(l-2), -1]$ must hold. If the offset is invalid, the state of the returned element will be undefined.

$$\dot{k}(n_{offset}) = \frac{0.5 \cdot k(n_{offset}+1) - 0.5 \cdot k(n_{offset}-1)}{\Delta t_{sample}} \tag{C.13}$$

`#getAccCD(offset : int32) : DATATYPE`
returns acceleration at the time step corresponding to `offset` calculated as Central Difference. $n_{offset} \in [-(l-2), -1]$ must hold. If the offset is invalid, the state of the returned element will be undefined.

$$\ddot{k}(n_{offset}) = \frac{k(n_{offset}+1) - 2 \cdot k(n_{offset}) + k(n_{offset}-1)}{(\Delta t_{sample})^2} \tag{C.14}$$

`#getJerkCD(offset : int32) : DATATYPE`
returns jerk at the time step corresponding to `offset` calculated as Central Difference. $n_{offset} \in [-(l-3), -2]$ must hold. If the offset is invalid, the state of the returned element will be undefined.

$$\dddot{k}(n_{offset}) = \frac{0.5 \cdot k(n_{offset}+2) - k(n_{offset}+1) + k(n_{offset}-1) - 0.5 \cdot k(n_{offset}-2)}{(\Delta t_{sample})^3} \tag{C.15}$$

`#getSpeedFD(offset : int32) : DATATYPE`
returns speed at the time step corresponding to `offset` calculated as Forward Difference. $n_{offset} \in [-(l-1), -1]$ must hold. If the offset is invalid, the state of the returned element will be undefined.

$$\dot{k}(n_{offset}) = \frac{k(n_{offset}+1) - k(n_{offset})}{\Delta t_{sample}} \tag{C.16}$$

`#getAccFD(offset : int32) : DATATYPE`
returns acceleration at the time step corresponding to `offset` calculated as Forward Difference. $n_{offset} \in [-(l-1), -2]$ must hold. If the offset is invalid, the state of the returned element will be undefined.

$$\ddot{k}(n_{offset}) = \frac{k(n_{offset}+2) - 2 \cdot k(n_{offset}+1) + k(n_{offset})}{(\Delta t_{sample})^2} \tag{C.17}$$

`#getJerkFD(offset : int32) : DATATYPE`
returns jerk at the time step corresponding to `offset` calculated as Forward Difference. $n_{offset} \in [-(l-1), -3]$ must hold. If the offset is invalid, the state of the returned element will be undefined.

$$\dddot{k}(n_{offset}) = \frac{k(n_{offset}+3) - 3 \cdot k(n_{offset}+2) + 3 \cdot k(n_{offset}+1) - k(n_{offset})}{(\Delta t_{sample})^3} \tag{C.18}$$

Using these functions, the following public methods allow to obtain speed, acceleration and jerk.

`+getSpeed(timestep : uint32) : DATATYPE`
returns the speed associated with the time step provided as argument. If the number of elements in the FIFO is smaller than 2, calculation isn't possible and the returned value is undefined. If the number of elements equals 2, there are two cases. If the argument equals the current time step, speed is calculated using Backward Difference, i.e. `getSpeedBD(offset : int32)`. If the argument equals the time step before the current one, Forward Difference is used, i.e. `getSpeedFD(offset : int32)`. If the number of elements is at least 3, there are three cases. If the argument equals the current time step, speed is calculated using Backward Difference, i.e. `getSpeedBD(offset : int32)`. If the argument equals the time step associated with the oldest trajectory element, Forward Difference is used, i.e. `getSpeedFD(offset : int32)`. In all other cases, Central Difference is used, i.e. `getSpeedCD(offset : int32)`. If the argument is invald, the returned value is undefined.

`+getAcc(timestep : uint32) : DATATYPE`
returns the acceleration associated with the time step provided as argument. If the number of elements in the FIFO is smaller than 3, calculation isn't possible and the returned value is undefined. If the number of elements is at least 3, there are three cases. If the argument equals the current time step, speed is calculated using Backward Difference, i.e. `getSpeedBD(offset : int32)`. If the argument equals the time step associated with the oldest trajectory element, Forward Difference is used, i.e. `getSpeedFD(offset : int32)`. In all other cases, Central Difference is used, i.e. `getSpeedCD(offset : int32)`. If the argument is invalid, the returned value is undefined.

`+getJerk(timestep : uint32) : DATATYPE`
returns the jerk associated with the time step provided as argument. If the number of elements in the FIFO is smaller than 4, calculation isn't possible and the returned value is undefined. If the number of elements equals 4, there are two cases. If the argument equals the current time step, speed is calculated using Backward Difference, i.e. `getSpeedBD(offset : int32)`. If the argument equals the time step before the current one, Forward Difference is used, i.e. `getSpeedFD(offset : int32)`. If the number of elements is at least 5, there are three cases. If the argument equals the current time step, speed is calculated using Backward Difference, i.e. `getSpeedBD(offset : int32)`. If the argument equals the time step associated with the oldest trajectory element, Forward Difference is used, i.e. `getSpeedFD(offset : int32)`. In all other cases, Central Difference is used, i.e. `getSpeedCD(offset : int32)`. If the argument is invald, the returned value is undefined.

`getNextSpeed(...)` behaves similarly to `getSpeed(...)` except that it takes `next_value` as current argument. Although `next_value` has not been pushed in the FIFO. the method does "as if". Such it is possible to determined speed associated with the next future (and maybe estimated) trajectory element. It is used to check if the next trajectory element ot be commanded violates speed or acceleration restrictions. In the same fashion, `getNextAcc(...)` corresponds to `getAcc(...)`, and `getNextJerk(...)` corresponds to `getJerk(...)`.

Current time step can be read by `getCurrTimestep()`. `getOldestTimestep()` returns the time step associated with the oldest element in the FIFO. `+operator[](...)` allows access to the individual elements in the FIFO. The argument is the absolute time step the element to be accessed is associated with. `numElements()` returns the number of elements in the FIFO.

In fact, the class' functionality exceeds the needs of RemoteSim as they are at the end of the developement process. However, the author decided to leave this functionality in case the code is used independently from RemoteSim in another project.

Context: The class `CAtomicTrajectoryGen` uses `CTrajectoryLog<double,20>` to caclulation current speed, get current position and time step for interpolation. The class `CQuatTrajectoryLog` is derived from `CTrajectoryLog`. It is used as `CQuatTrajectoryLog<CQuaternion,20>` in `CQuatTrajectoryGen` similarly, i.e. for calculating current angular speed, getting current position and time step.

C.41. class CTransceiver

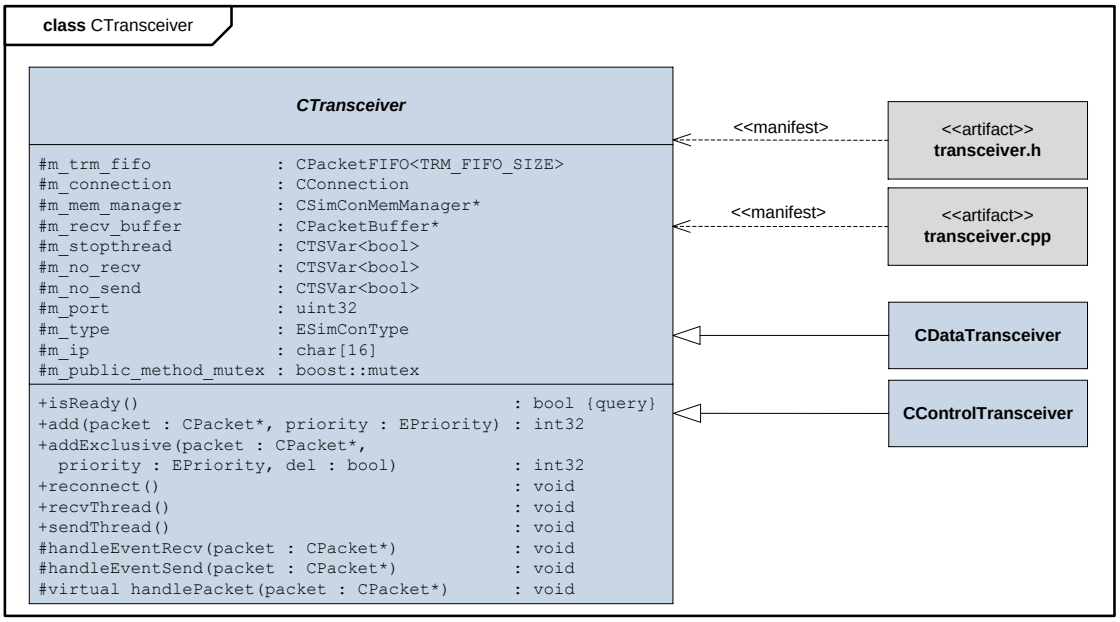

Figure C.46.: UML Class Diagram of class CTransceiver

Description: `CTransceiver` is an abstract `class`. It combines elements that are commonly needed by the derived `classes` `CDataTransceiver` and `CControlTransceiver`. It contains an instance of `CConnection` to realize a TCP/IP connection. An instance of `CPacketFIFO` buffers SCP packets to be transmitted. The methods `recvThread()` and `sendThread()`, which are executed as separate threads, handle receipt and transmission of packets. `isReady()` indicates whether the `class` is ready for use (connection established). `add(...)` and `addExclusive(...)` represent the according methods of `CPacketFIFO`. `reconnect()` represens the according method of `CConnection`.

Context: `CTransceiver` is an abstract `class` and is not instanciated. The `classes` `CDataTransceiver` and `CControlTransceiver` are derived from that `class`.

C.42. class CVec3D

Description: This `class` implements a three-dimensional vector with its related operators. Note the difference between `normalize()` and `normalized()`. While the former changes the state of the object, the latter merely returns a normalized intance of the object. Alle other methods are self-explanatory. Binary operators are implemented as separate functions and not as member functions.

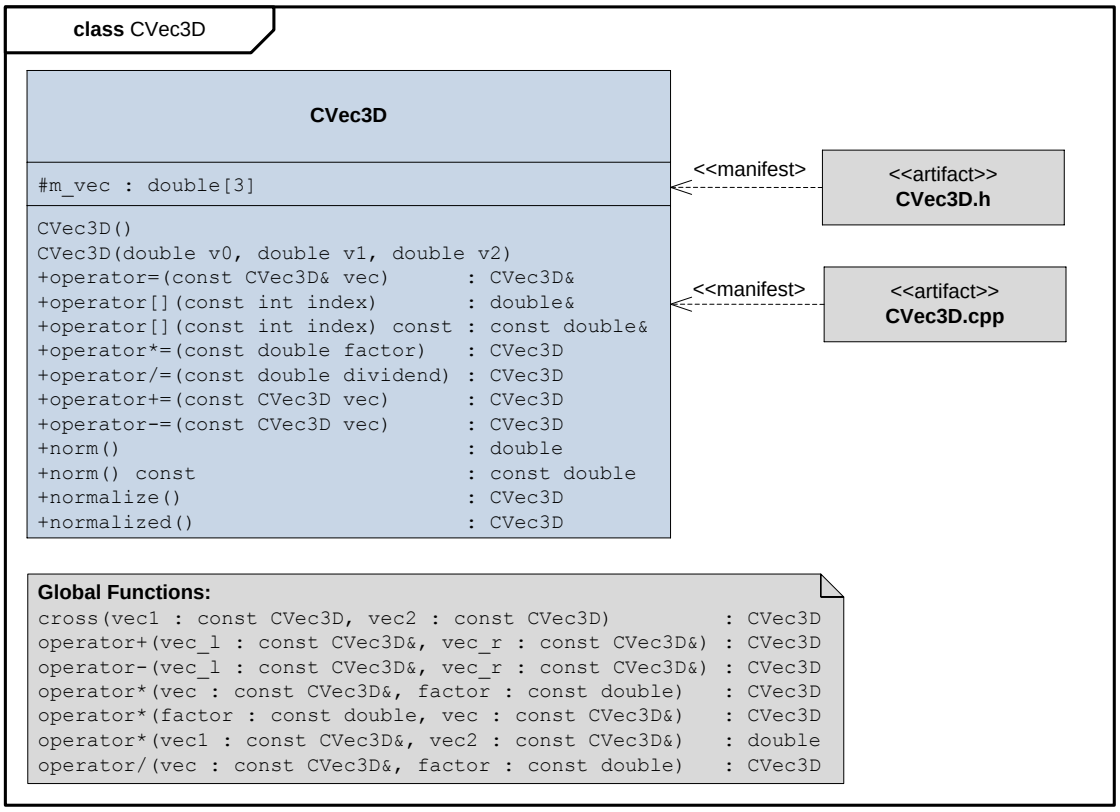

Figure C.47.: UML Class Diagram of class CVec3D

Context: The `class CVec3D` is used at numerous points as position, speed and angular velocity vector.

C.43. struct SFormationState

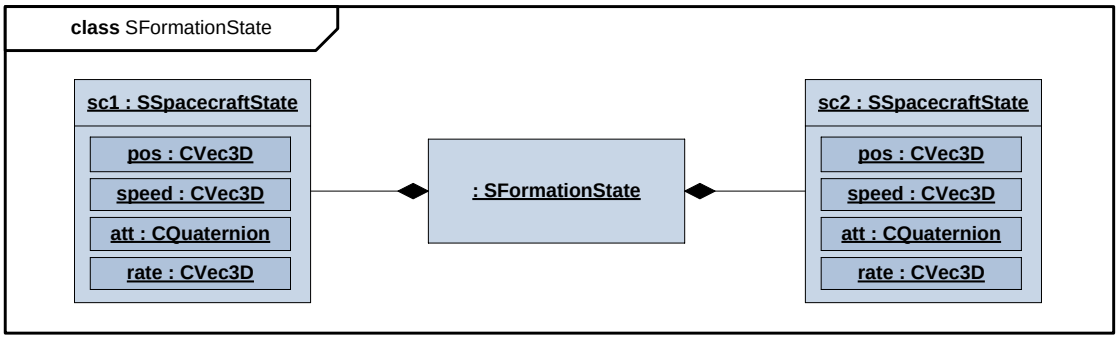

Figure C.48.: UML Class Diagram of struct SFormationState

Description: The `struct SFormationState` wraps up two instances of `SSpacecraftState`, each representing the state of one S/C, consisting of position, speed, attitude and rate (angular velocity).
Context: `SFormationState` is used at various points not only with RemoteSim-Client, but also with RemoteSim-Server.

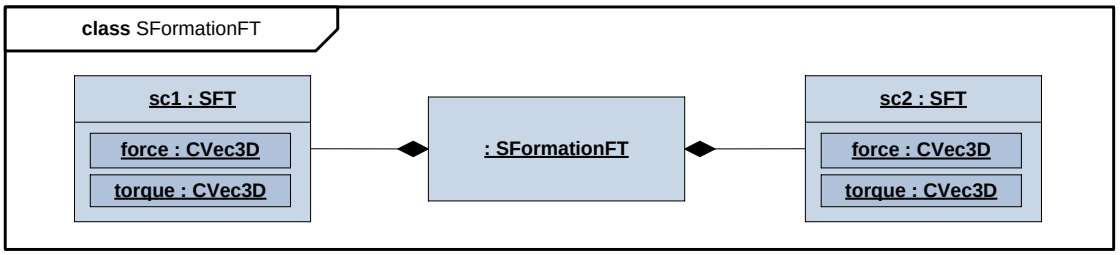

Figure C.49.: UML Class Diagram of struct SFormationFT

C.44. struct SFormationFT

Description: The `struct SFormationFT` wraps up two instances of `SFT`, each representing force and torque acting on one S/C.

Context: `SFormationFT` is used at various points not only with RemoteSim-Client, but also with RemoteSim-Server.

D. RemoteSim User's Guide

D.1. Building RemoteSim

There is more than one way to build RemoteSim-Server and RemoteSim-Client. The author used the `mex` tool provided by Matlab. The dvd of this thesis includes two Matlab scripts: `compileRemoteSimServer.m` and `compileRemoteSimClient.m`. Preserving the file structure on the dvd, the location of the directory "/include" must be altered in the files appropriately (toolbox_dir). Then, execution of the scripts places the `mex` files in "/include/eposmex". This directory can also be altered as needed. Note that for successfull compilation, the file `mexopts.bat` (part of simulink) must be edited: The `INCLUDE` section must contain "'toolbox_dir'/include/remotesim", "'toolbox_dir'/include/simcon" and "'toolbox_dir'/include/misc". Moreover, in the `LIB` section the location of the boost::thread library, which is required for RemoteSim, must be given. The version of this library depends on the platform and the compiler used by Matlab. For compilation with the Real-Time Workshop, the aforementioned directories and `RemoteSimLibrary.mdl` should be integrated in the EPOS toolbox.

D.2. Modelling a RemoteSim-Client Simulation

Building a RemoteSim-Client simulation is very simple. Besides RemoteSim-Client, an enabled subsystem is needed in the Simulink model. The actual simulation the user wishes to run is to be placed in this enabled subsystem. RemoteSim-Client output "Simulation Enable" is to be connected to the enable input of the enabled subsystem with a unit-delay in-between. This unit-delay avoids an algebraic loop which would occur inevitably. RemoteSim-Client accounts for this unit-delay and activates the enable output one step ahead, so that the subsystem is enabled at the right time. One important note: For the model located in the enabled subsystem, the execution time is different from the Simulink simulation time, since the simulation will already be running for some time at the point when the subsystem is enabled. This must be considered when designing the model to be put into the subsystem. RemoteSim-Client inputs have to be connected to the user models outputs, in general the outputs of the enabled subsystem. These are chaser and target states for trajectory commanding or chaser and target force/torque for force/torque commanding. The other RemoteSim-Client outputs can be used for monitoring bur are not vital for the simulation. Parameters have to be set according to Tab.2.4.

D.3. Modelling a RemoteSim-Server Simulation

The RemoteSim-Server outputs "Chaser Command Position", "Chaser Command Attitude", "Target Command Position" and "Target Command Attitude" are to be connected to the inputs of the EPOS CMD interface. If needed, additional modelling is needed to calculate a POV from these values for the CMD interface. The CMD interfaces outputs (robot states) should be connected to the RemoteSim-Server inputs "Chaser Current Position (CLW)", "Chaser Current Attitude (CLW)", "Target Current Position (CLW)" and "Target Current Attitude (CLW)" so that RemoteSim-Client can use this data. However, these inputs are not required for making RemoteSim work. The enable output of the EPOS CMD interface is to be connected to the RemoteSim-Server input "CMD Interface Enable" with a unit-delay in-between to avoid an algebraic loop. Note that this unit-delay was already included in the CMD interface at the time this thesis was finished. If RemoteSim is used with force/torque commanding, another component is needed: the integrator. It must be placed into a enabled subsystem with similar consequences as outlined in Sec.D.2. The RemoteSim-Server output "Integrator Enable" is to be connected to the enable input of the subsystem, again with a unit-delay.

The inputs of the integrator have to be connected to RemoteSim-Server outputs "Chaser Integrator Force", "Chaser Integrator Torque", "Target Integrator Force" and "Target Integrator Torque". Moreover, the integrator requires initial conditions which are supplied by outputs "Chaser Integrator Initial State" and "Target Integrator Initial State". The outputs of the integrator are expected to comprise chaser and target states. With a unit-delay for each signal, they must be connected to "Chaser Integrator Position (U)", "Chaser Integrator Speed (U)", "Chaser Integrator Attitude (U)", "Chaser Integrator Rate (U)", "Target Integrator Position (U)", "Target Integrator Speed (U)", "Target Integrator Attitude (U)" and "Target Integrator Rate (U)". The RemoteSim-Server input "Reset" needs to be connected such that the user can change its value from 0 to 1 and back again for at least one time step. This can easily be achieved by using a manual switch. RemoteSim-Server outputs not mentioned here are for monitoring only. Parameters have to be set according to Tab.2.9.

D.4. Simulation Procedure

In general, it doesn't matter if RemoteSim-Client simulation or RemoteSim-Server simulation is started first. However, from a practical perspective, it is recommended to begin with the RemoteSim-Server simulation. As soon as the RemoteSim-Server simulation is started, the procedure for synchronous commanding ("Sync", "Move to Start", "Confirm Sync", see Sec.1.2.4) must be carried out. Therafter, RemoteSim-Server is ready. The remote simulation can be started. From now on, initial trajectory is determined, the robots move accordingly and seamless transition to the simulation occurs. If the remote simulation is stopped or a CMD is delayed due to another reason, RemoteSim-Server returns to Safe-Slow-Down state and signals an error at the output "Error". At this point, the RemoteSim-Server simulation does not have to be restarted. Instead, a 1 for at least one time step (and then 0) at the input "Reset" confirms the error and makes RemoteSim-Server ready for another simulation. The RemoteSim-Client simulation can be started anew.

E. Files

Table E.1.: Header Files.

File	Category	Content
atomic_trajectory_gen.h	RemoteSim	Declaration of class CAtomicTrajectoryGen. Definition of enum EATrajGenError.
cart_trajectory_gen.h	RemoteSim	Declaration of class CCartTrajectoryGen and enum ECTrajGenError.
client_step_timer.h	RemoteSim	Declaration of class CClientStepTimer.
connection.h	SimCon	Declaration of class CConnection.
container_list.h	SimCon	Definition of template CContainerList<LIST_ELEMENT_CLASS> and definition of associated error constants.
control_transceiver.h	SimCon	Declaration of class CControlTransceiver.
conversion.h	SimCon	Declaration of class CConversion and definition of related constants. Definition of enum EEndien.
cosy_converter.h	RemoteSim	Declaration of class CCosyConverter.
data_transceiver.h	SimCon	Declaration of class CDataTransceiver.
delaymanager.h	SimCon	Declaration of class CDelayManager and class CDelay. Definition of constant DELAY_FIFO_SIZE.
formation_state_-trajectory_log.h	RemoteSim	Definition of template CFormationStateTrajectoryLog<LOG_FIFO_SIZE> and definition of constant LOG_FIFO_SIZE.
hermite_quaternion.h	RemoteSim	Declaration of class CHermiteQuaternion. Definition of enum EHQuatError.
lin_equs_sys.h	RemoteSim	Declaration of class CLinEqusSys and associated error constants.
object_repository.h	SimCon	Declaration of class CObjectRepository.
packet.h	SimCon	Declaration of class CPacket.
packetbuffer.h	SimCon	Declaration of class CPacketBuffer.
packet_fifo.h	SimCon	Definition of template CPacketFIFO<MAX_SIZE, enum EPriority and related constants.
packet_header_spec.h	SimCon	Specification of SCP header structure.

Continued on next page

Table E.1 – *Continued from previous page*

File	Category	Content
packet_types.h	SimCon	Definition of enum EPacketType, enum EHeaderValueDatatype, enum EPacketError and struct SHeaderValue.
quaternion.h	RemoteSim	Declaration of class CQuaternion. Definition of constant CONSTANT_PI .
quat_trajectory.h	RemoteSim	Declaration of class CQuatTrajectory. Definition of enum EQuatTrajError.
quat_trajectory_gen.h	RemoteSim	Declaration of class CQuatTrajectoryGen. Definition of enum EQTrajGenError.
quat_trajectory_log.h	RemoteSim	Definition of template CQuatTrajectoryLog<FIFO_SIZE.
remotesim_block.h	RemoteSim	Definition of struct SRemoteSimServerBlock and struct SRemoteSimClientBlock.
remotesim_client.h	RemoteSim	Declaration of class CRemoteSimClient.
remotesim_server.h	RemoteSim	Declaration of class CRemoteSimServer. Definition of constant INIT_DELAY.
remotesim_sfun_spec.h	RemoteSim	Specification of RSP packet structure.
remotesim_types.h	RemoteSim	Definition of enum ECoSy, enum ECMD, enum EState, enum EMode, enum EError and primitive data types.
ringqueue.h	SimCon	Declaration of class CRingQueue and definition of constant RINGQUEUE_INVALID_INDEX.
robot.h	RemoteSim	Declaration of class CRobot and definition of associated error constants.
simcon.h	SimCon	Declaration of class CSimCon and definition of related error constants.
simcon_memmanager.h	SimCon	Declaration of class CMemoryBlock, class CSimConMemManager, class CSimConMemManagedClass and related error constants.
simcon_types.h	SimCon	Definition of enum ESimConType, enum ESimConDataType and primitive data types.
static_list.h	SimCon	Definition of template CStaticList<DATATYPE,MAX_SIZE, template CStaticList<DATATYPE> and constants STATIC_LIST_FULL and STATIC_LIST_INVALID_INDEX.
static_stack.h	SimCon	Definition of template CStaticStack<DATATYPE,NUM_ELEMENTS>.
step_counter.h	RemoteSim	Declaration of class CStepCounter.
step_timer.h	RemoteSim	Declaration of class CStepTimer.
thisthat.h	SimCon	Declaration of function getTime() and function getPosixTime().

Continued on next page

Table E.1 – *Continued from previous page*

File	Category	Content
trajectory_gen.h	RemoteSim	Declaration of `class CTrajectoryGen` and associated error constants.
trajectory_log.h	RemoteSim	Definition of `template CTrajectoryLog<DATATYPE,FIFO_SIZE>`.
trajectory.h	RemoteSim	Declaration of `class CTrajectory`. Definition of constants `TIME_INTERVAL_INVALID` and `TRAJECTORY_ERROR`.
transceiver.h	SimCon	Declaration of `class CTransceiver`. Definition of constants `TRM_FIFO_SIZE` and `TRANSCEIVER_INPUT_FIFO_FULL`.
vec_3D.h	RemoteSim	Declaration of `class CVec3D`.

Table E.2.: Source Files.

File	Category	`class` Definition
atomic_trajectory_gen.cpp	RemoteSim	`CAtomicTrajectoryGen`
cart_trajectory_gen.cpp	RemoteSim	`CCartTrajectoryGen`
client_step_timer.cpp	RemoteSim	`CClientStepTimer`
connection.cpp	SimCon	`CConnection`
control_transceiver.cpp	SimCon	`CControlTransceiver`
conversion.cpp	SimCon	`CConversion`
cosy_converter.cpp	RemoteSim	`CCosyConverter`
data_transceiver.cpp	SimCon	`CDataTransceiver`
delaymanager.cpp	SimCon	`CDelayManager`, `CDelay`
hermite_quaternion.cpp	RemoteSim	`CHermiteQuaternion`
lin_equs_sys.cpp	RemoteSim	`CLinEqusSys`
object_repository.cpp	SimCon	`CObjectRepository`
packet.cpp	SimCon	`CPacket`
packetbuffer.cpp	SimCon	`CPacketBuffer`
quaternion.cpp	RemoteSim	`CQuaternion`
quat_trajectory.cpp	RemoteSim	`CQuatTrajectory`
quat_trajectory_gen.cpp	RemoteSim	`CQuatTrajectoryGen`
remotesim_client.cpp	RemoteSim	`CRemoteSimClient`

Continued on next page

Table E.2 – *Continued from previous page*

File	Category	Content
remotesim_server.cpp	RemoteSim	`CRemoteSimServer`
ringqueue.cpp	SimCon	`CRingQueue`
robot.cpp	RemoteSim	`CRobot`
simcon.cpp	SimCon	`CSimCon`
simcon_memmanager.cpp	SimCon	`CMemoryBlock`, `CSimConMemManager`, `CSimConMemManagedClass`
step_counter.cpp	RemoteSim	`CStepCounter`
step_timer.cpp	RemoteSim	`CStepTimer`
thisthat.cpp	SimCon	`getTime()`, `getPosixTime()`
trajectory_gen.cpp	RemoteSim	`CTrajectoryGen`
trajectory.cpp	RemoteSim	`CTrajectory`
transceiver.cpp	SimCon	`CTransceiver`
vec_3D.cpp	RemoteSim	`CVec3D`

Table E.3.: Other Files.

File	Description
remotesim_server_block.tlc	TLC (Target Language Compiler) file for Simulink Real-Time Workshop.
RemoteSimLibrary.mdl	Simulink library containing RemoteSim-Client and RemoteSim-Server block.

Bibliography

[1] Boge T., Wimmer T., Ma O., Zebenay M., *EPOS-A Robotics-Based Hardware-in-the-Loop Simulator for Simulating Satellite RvD Operations*, The 10th International Symposium on Artificial Intelligence, Robotics and Automation in Space, Sapporo, Japan, 2010

[2] Boge T., Wimmer T., Ma O., Tzschicholz T, *EPOS - Using Robotics for RvD Simulation of On-Orbit Servicing Missions*, AIAA Modeling and Simulation Technologies Conference, Toronto, Canada, 2010

[3] Gaias G., D'Amico, Boge T, *Hardware-in-the-loop Multi-satellite Simulator for Proximity Operations*, 11th Int. WS on Simulation & EGSE facilities for Space Programmes; SESP 2010, Noordwijk, Netherlands, 2010

[4] Tzschichholz T., Boge Th., *GNC Systems Development in Conjunction with a RVD Hardware-in-the-loop Simulator*, 4th International Conference on Astrodynamics Tools and Techniques, Madrid, 2010

[5] Boge T., Rupp Th., Landzettel K., Wimmer T., Mietner Ch., Bosse J., Thaler B, *Hardware in the Loop Simulator für Rendezvous und Docking Manöver*, DGLR Jahrestagung, Aachen, 2009

[6] Robo-Technology GmbH, *EPOS Facility Manual*

[7] James F. Kurose, Keith W. Ross, *Computer Networking - A Top-Down Approach*, Fourth Edition, Pearson International Edition, 2008

[8] D'Amico S., *Autonomous formation flying in low earth orbit*, PhD thesis, Technical University of Delft, 2010

[9] S. D'Amico, R. Larsson, *Navigation and Control of the PRISMA formation: In-Orbit Experience*, 18th IFAC World Congress, Milano, Italy, 2011

[10] Jack B. Kuipers, *Quaternions and Rotation Sequences - A Primer with Applications to Orbits, Aerospace and Virtual Reality*, Princeton University Press, 1999

[11] Gross, Hauger, Schröder, Wall, *Technische Mechanik 3*, 9. Auflage, Springer, Berlin Heidelberg, 2006

[12] Bengt Fornberg, *Generation of Finite Difference Formulas on Arbitrarily Spaced Grids*, Mathematics of Computation, Volume 51, Number 184, Pages 699-706, October 1988

[13] J. S. Ardaens, S. D'Amico, *Formation Flying Testbed*, TN 09-01, Version 0.2, DLR, 26.02.2009

[14] Montenbruck O., Nortier B., Mostert S., *A Miniature GPS Receiver for Precise Orbit Determination of the SUNSAT2004 Micro-Satellite*, ION Natinal Technical Meeting, San Diego, California, 26-28 Januar 2004

[15] L. Rade, B. Westergren, *Springers Mathematische Formeln*, Springer, 2000

[16] Kim, Kim, Shin, *A General Construction Scheme for Unit Quaternion Curves with Simple High Order Derivatives*

[17] Mathworks, *Simulink Online Help*, http://www.mathworks.de/help/toolbox/simulink/index.html

[18] Wikipedia, *Quartz clock*, http://en.wikipedia.org/wiki/Quartz_clock