ESROCOS

DETAILED DESIGN DOCUMENT

ESROCOS_D3.1

Due date of deliverable:	31-07-2017
Start date of project:	01-11-2016
Duration:	27 months
Topic:	COMPET-4-2016 Building Block a) Space Robot Control Operating System
Work package:	3100, 3200
Lead partner for this deliverable:	ISAE

This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement No 730080.

Dissemination Level

PU	Public	**X**
CO-1	Confidential, restricted under conditions set out in Model Grant Agreement. Version providing the PSA will all the information required to perform its assessment.	
CO-2	Confidential, restricted under conditions set out in Model Grant Agreement. Version providing the PSA and the other operational grant the information required for the integration of all the building blocks and the continuity of the Strategic Research Cluster	

Prepared by: ESROCOS team

Approved by: ISAE

Authorized by: Jérôme Hugues

Code: ESROCOS_D3.1

Version: 1.1

Date: 19/04/2018

DOCUMENT STATUS SHEET

Version	Date	Pages	Changes
1.0	31/01/2018	132	First issue of the document.
1.1	19/04/2018	139	Update with CDR RIDs: CDR_RID_01, CDR_RID_02 and CDR_RID_04 to CDR_RID_12.

TABLE OF CONTENTS

LIST OF TABLES AND FIGURES

1. INTRODUCTION

1.1. PURPOSE

The PERASPERA OG1 activity is devoted to the design of a Robot Control Operating Software (RCOS) that can provide adequate features and performance with space-grade Reliability, Availability, Maintainability and Safety (RAMS) properties. The goal of the ESROCOS project is to provide an open source framework which can assist in the development of flight software for space robots. By providing an open standard which can be used by research labs and industry, it is expected that the Technology Readiness Level (TRL) can be raised more efficiently, and vendor lock-in through proprietary environments can be reduced. Current state-of-the-art robotic frameworks are already addressing some of these key aspects, but mostly fail to deliver the degree of quality expected in the space environment. In the industrial robotics world, manufacturers of robots realise their RCOS by complementing commercial real-time operating systems, with proprietary libraries implementing the extra functions.

The Detailed Design Document presents the architecture of the system and the static and dynamic architecture of its main components.

1.2. SCOPE

This document is an outcome of the WP 3100 "RCOS Prototyping" and 3200 "Definition and Design of Reference Implementations" of the ESROCOS activity. These WPs establish the detailed design of the ESROCOS framework. The ESROCOS framework is a set of tools and software components that support the development of robotics applications with demanding RAMS requirements. It consists of Robot Control Operating System (RCOS) components, and RCOS Development Environment (RDEV) tools.

In this document we detail the design of the software components that constitute the ESROCOS framework and that were identified in [AD.6]. In the ESROCOS project some requirements are covered by components developed from scratch, while some others will be covered by the integration of existing software. The document describes the design of each component according to the scope of the work foreseen in the activity. This means that newly developed components will be described globally, while for existing components the design will focus on their integration in the framework.

For the RCOS (runtime) components, the detailed design covers the static and dynamic architecture of each component. For the RDEV (development) components, the document focuses on the static design.

1.3. CONTENTS

This document contains the following sections:

- Section 1: Introduction.
- Section 2: Applicable and reference documents. Lists of documents that are relevant to the structure and contents of this document.
- Section 3: Terms, definitions and abbreviated terms. List of terms and definitions that harmonize the nomenclature used providing the clarifications for the correct understanding of the terms.
- Section 4: Software Overview. Presents the overall structure of the ESROCOS framework and identifies its constituent software products.
- Section 5: Software Component Design. Provides the preliminary design of each of the software products in the ESROCOS framework.

2. REFERENCE AND APPLICABLE DOCUMENTS

2.1. APPLICABLE DOCUMENTS

The following is the set of documents that are applicable:

Ref.	Title	Date
[AD.1]	Peraspera: D3.1 Compendium of SRC Activities (for call 1)	
[AD.2]	Guidelines for strategic research cluster on space robotics technologies horizon 2020 space call 2016	
[AD.3]	Call: H2020-COMPET-2016 ESROCOS Proposal. Proposal no. 730080	03/03/2016
[AD.4]	ESROCOS System Requirements (D1.2) issue 1.1. GMV 20172/17 V2/17	23/06/2017
[AD.5]	ESROCOS Interface Control Document (D2.3) issue 2.0. GMV 20177/17 V2/17	23/06/2017
[AD.6]	ESROCOS Preliminary Design Document (D2.2) issue 1.1. GMV 22280/17 V2/17	06/09/2017

Table 2-1. Applicable documents

2.2. REFERENCE DOCUMENTS

The following is the set of documents referenced or useful for understanding the contents of this report:

Ref.	Title	Date
[RD.1]	ESROCOS Technology Review (D.1.1). GMV 20095/17 V1/17	17/01/2017
[RD.2]	G. Visentin, "The case for an RCOS at ESA", ASTRA 2015 – RCOS forum	2015
[RD.3]	The Robot Operating System (ROS). Online at: http://www.ros.org/	-
[RD.4]	The Robot Construction Kit (ROCK). Online at: http://rock-robotics.org	-
[RD.5]	The Open Robot Control Software (OROCOS). Online at: http://www.orocos.org/	-
[RD.6]	TASTE. Online at: http://TASTE.tuxfamily.org/wiki/index.php?title=Main_Page	-
[RD.7]	Behaviour, Interaction, Priority (BIP). Online at: http://wwwverimag.imag.fr/Rigorous-Design-of-Component Based.html	-
[RD.8]	"A Verifiable and Correct-by-Construction Controller for Robot Functional Levels", Bensalem et al, Journal of Software Engineering for Robotics, September 2011	2011
[RD.9]	ASN.1 standard. Online at: http://www.itu.int/en/ITU-T/asn1/Pages/asn1_project.aspx	-
[RD.10]	AADL. Online at: http://www.aadl.info/aadl/currentsite/	-
[RD.11]	"TASTE: An open-source tool-chain for embedded system and software development", Maxime Perrotin, Eric Conquet, Julien Delange, Thanassis Tsiodras	2012
[RD.12]	M. Perrotin, E. Conquet, J. Delande, A. Schiele, T. Tsiodras, "TASTE: A real-time software engineering tool-chain Overview, status, and future"	2011
[RD.13]	M. Perrotin, "Support for device drivers in TASTE"	2010
[RD.14]	Perrotin, M., Conquet, E., Dissaux, P., Tsiodras, T., Hugues, H.: The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software, ERTS'2010, Toulouse	2010
[RD.15]	The ITU-T SDL Standard. Online at: http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf	-
[RD.16]	European Cooperation for Space Standardization. Space Engineering — Telemetry and telecommand packet utilization (PUS) Services. ECSS-E-ST-70-41C, Apr. 15th, 2016	Apr. 2016
[RD.17]	European Cooperation for Space Standardization. Space Engineering – Software. ECSS-E-ST-40C, Mar. 6th, 2009.	Mar. 2009
[RD.18]	The OpenCV library. Online at: http://opencv.org/	-
[RD.19]	The Eigen template library for linear algebra. Online at: http://eigen.tuxfamily.org/	-
[RD.20]	ROCK Stream aligner. Online at http://rock-robotics.org/stable/documentation/data_processing/stream_aligner.html	-

Ref.	Title	Date
[RD.21]	ROCK Transformer. Online at http://rock-robotics.org/stable/documentation/data_processing/transformer.html	-
[RD.22]	Collada. Online at: https://www.khronos.org/collada/	-
[RD.23]	Khronos Group. COLLADA – Digital Asset Schema Release 1.5.0 Specification. Online at: https://www.khronos.org/files/collada_spec_1_5.pdf	Apr. 2008
[RD.24]	European Cooperation for Space Standardization. Space Engineering – Spacecraft on-board control procedures. ECSS-E-ST-70-01C, Apr. 16th, 2010	Apr. 2016
[RD.25]	P. E. Bulychev, A. David, K. G. Larsen, A. Legay, G. Li, and D. B. Poulsen. Rewrite-based statistical model checking of wmtl. RV, 7687:260-275, 2012.	2012
[RD.26]	A. Wald. Sequential tests of statistical hypotheses. Annals of Mathematical Statistics, 16(2):117-186, 1945.	1945
[RD.27]	H. L. S. Younes. Verification and Planning for Stochastic Processes with Asynchronous Events. PhD thesis, Carnegie Mellon, 2005.	2005
[RD.28]	T. Herault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate Probabilistic Model Checking. In International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI'04, pages 73-84, January 2004.	2004
[RD.29]	S. Bensalem, B. Delahaye, and A. Legay. Statistical model checking: Present and future. In RV, volume 6418 of LNCS. Springer, 2010.	2010
[RD.30]	W. Hoeding. Probability inequalities. Journal of the American Statistical Association, 58:13-30, 1963.	1963
[RD.31]	A. Nouri, B. L. Mediouni, M. Bozga, J. Combaz, A. Legay, and S. Bensalem. Performance evaluation of stochastic real-time systems with the sbip framework. Technical Report TR-2017-6, Verimag Research Report, 2017.	2017
[RD.32]	R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci., 126(2):183–235*, 1994.	1994
[RD.33]	P. Dissaux, P. Farail. Model Verification: Return of Experience. In Embedded Real-Time Software and Systems (ERTS 2014) Proceedings.	2014
[RD.34]	P. Dissaux, B. Hall. Merging and Processing Heterogeneous Models. In Embedded Real-Time Software and Systems (ERTS 2016) Proceedings.	2016
[RD.35]	MicroPython, Online at: https://micropython.org	-
[RD.36]	D. George et al. Porting of MicroPython to LEON platforms (Executive Summary) Rev.2. April 2016	Apr. 2016
[RD.37]	H. Bruyninckx, E. Scioni, N.Hübel. "Composable control stacks in component-based cyber-physical system platforms – With application to robotics system-of-systems". KU Leuven, Belgium. Online at: https://people.mech.kuleuven.be/~bruyninc/tmp/modelling-for-motion-stack.pdf	Mar. 2018
[RD.38]	I. Dragomir. "Formalization of TASTE Designs as Networks of Timed Automata". Verimag, Universite Grenoble Alpes. Online at: https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/bip/TASTE2BIP.git	Dec. 2017

Table 2-2. Reference documents

3. TERMS DEFINITIONS AND ABBREVIATED TERMS

3.1. DEFINITIONS

Concepts and terms used in this document and needing a definition are included in the following table:

Table 3-1. Definitions

Concept / Term	Definition
-	-

3.2. ACRONYMS

Acronyms used in this document and needing a definition are included in the following table:

Table 3-2. Acronyms

Acronym	Definition
AADL	Architecture Analysis and Design Language
ACK	Acknowledgement
ACN	ASN.1 Concrete Notation
AIR	ARINC 653 Interface in RTEMS
ANTLR	Another Tool for Language Recognition
API	Application Programming Interface
APID	Application Identifier
ARINC	Aeronautical Radio, Incorporated
ASN.1	Abstract Syntax Notation One
ASSERT	Automated proof-based System and Software Engineering for Real-Time applications
AST	Abstract Syntax Tree
BIP	Behaviour, Interaction, Priority
BLTL	Bounded Linear Temporal Logic
BSP	Board Support Package
CAN	Controller Area Network
CANBUS	Controller Area Network Bus
CCSDS	Consultative Committee for Space Data Systems
CDS	CCSDS Day Segmented
COLLADA	Collaborative Design Activity
CPU	Central Processing Unit
CUC	CCSDS Unsegmented Code
CV	Concurrency View
DFKI	Deutsches Forschungszentrum für Künstliche Intelligenz
DV	Deployment View
ECSS	European Cooperation for Space Standardization
EMF	Eclipse Modelling Framework
ERGO	European Robotic Goal-Oriented Autonomous Controller
ESA	European Space Agency

Acronym	Definition
ESOC	European Space Operations Centre
ESROCOS	European Space Robotics Control and Operating System
FDIR	Failure Detection, Isolation and Recovery
FHPE	Functional Hypervisor Partition Emulator
GCC	GNC Compiler Collection
GNU	GNU is Not UNIX
GPS	Global Positioning System
GR	Gaisler Research
GSL	GNU Scientific Library
GUI	Graphical User Interface
H2020	Horizon 2020
HAIR	Hypervisor emulator based on AIR
HDD	Hard Disk Drive
HLIM	Hardware Interface Model
HPE	Hypervisor Partition Emulator
HT	Hypothesis Testing
HW	Hardware
I/O	Input / Output
ID	Identifier
IDE	Integrated Development Environment
IDL	Interface Description Language
ILK	Intermediate Language for Kinematics
IMU	Inertial Measurement Unit
IO	Input / Output
ISAE	Institut Supérieur de l'Aéronautique et de l'Espace
IV	Interface View
JRE	Java Runtime Environment
JSON	Javascript Object Notation
KUL	Katholieke Universiteit Leuven
LIDAR	Light Detection and Ranging
LMP	Logic Model Processing
LTL	Linear Temporal Logic
MQ	Message Queue
MTL	Metric Temporal Logic
MVC	On-Board Control Procedure
NTP	Network Time Protocol
OBCP	On-Board Control Procedure
OCL	Object Constraint Language
OG	Operational Grant
OROCOS	The Open Robot Control Software
PB-LTL	Probabilistic Bounded Linear Temporal Logic

Acronym	Definition
PC	Personal Computer
PDD	Preliminary Design Document
PE	Processor Emulator
PESTIM	Probability Estimation
PI	Provided Interface
PIM	Partition Interference Model
PMK	Partition Management Kernel
PMON	Parameter Monitoring
POS	Partition Operating System
POSIX	Portable Operating System Interface Unix
PUS	Packet Utilization Standard
PWM	Pulse Width Modulation
QA	Quality Assurance
RAMS	Reliability, Availability, Maintainability and Safety
RCOS	Robot Control Operating System
RD	Reference Document
RDEV	RCOS Development Environment
RI	Required Interface
ROCK	Robot Construction Kit
ROS	Robot Operating System
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	Real-Time Operating System
RTT	Real-Time Toolkit
SAPI	Simulator API
SARGON	Space Automation and Robotics General Controller
SDL	Specification and Description Language
SI	International System of Units
SLIM	Software Interference Model
SMC	Statistical Model Checking
SMP2	Simulation Model Portability version 2
SOEM	Simple Open EtherCAT Master
SPARC	Scalable Processor Architecture
SPRT	Sequential Ratio Testing Procedure
SSD	Solid State Drive
SSP	Single Sampling Plan
SW	Software
TASTE	The ASSERT Set of Tools for Engineering
TBC	To Be Confirmed
TC	Telecommand
TCP	Transmission Control Protocol
TM	Telemetry

Acronym	Definition
TRL	Technology Readiness Level
TSP	Time and Space Partitioning
UGA	Université Grenoble Alpes
UML	Unified Modelling Language
URDF	Unified Robot Description Format
WP	Work Package
XML	Extensible Mark-up Language
YAML	YAML Ain't Mark-up Language

4. SOFTWARE OVERVIEW

ESROCOS is a framework for developing robot control software applications. It includes a set of tools that support different aspects of the development process, from architectural design to deployment and validation. In addition, it provides a set of core functions that are often used in robotics or space applications.

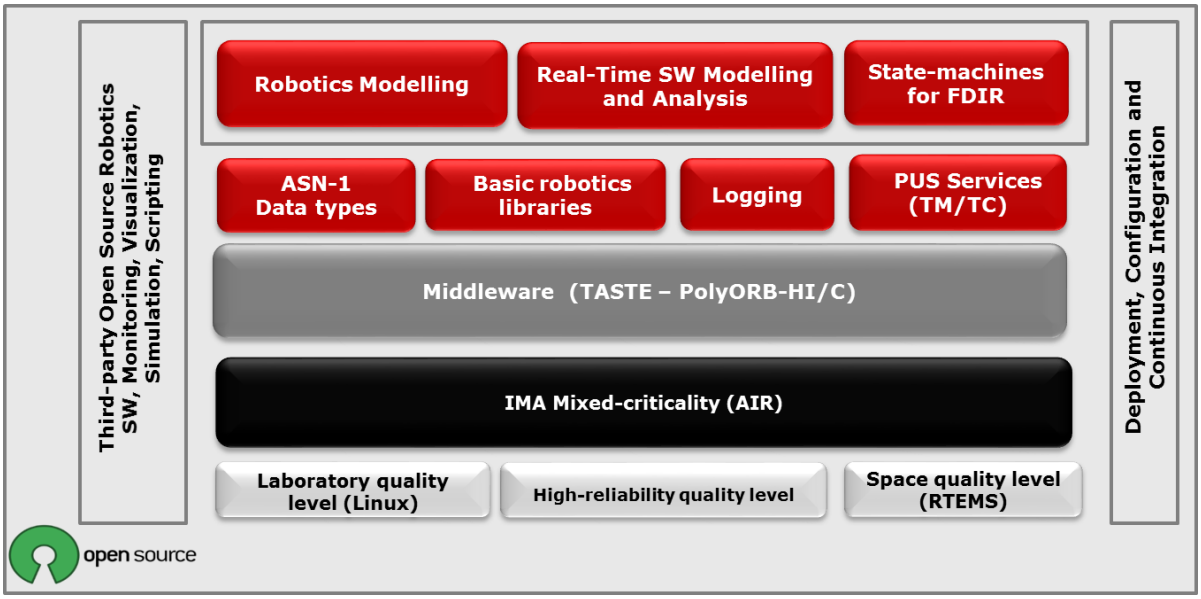


Figure 4-1. Overview of the components of ESROCOS

The ESROCOS framework is intended to support the development of software following the ECSS standards. It does not by itself cover all the development phases and verification steps, but it facilitates certain activities and ensures that the software built can be made compatible with the RAMS requirements of critical systems.

Figure 4-2 summarizes the main activities supported by the ESROCOS framework. The rounded white boxes indicate activities, grey rectangles denote software artefacts (models, source code, applications, etc.), and dashed boxes group related items. The software artefacts are either product of the activities (identified in italics), or directly provided by the framework as functional blocks to use in the activities. The figure illustrates how the activities, their products and the functional blocks are combined to form a consistent workflow for the production of distributed robot application software.

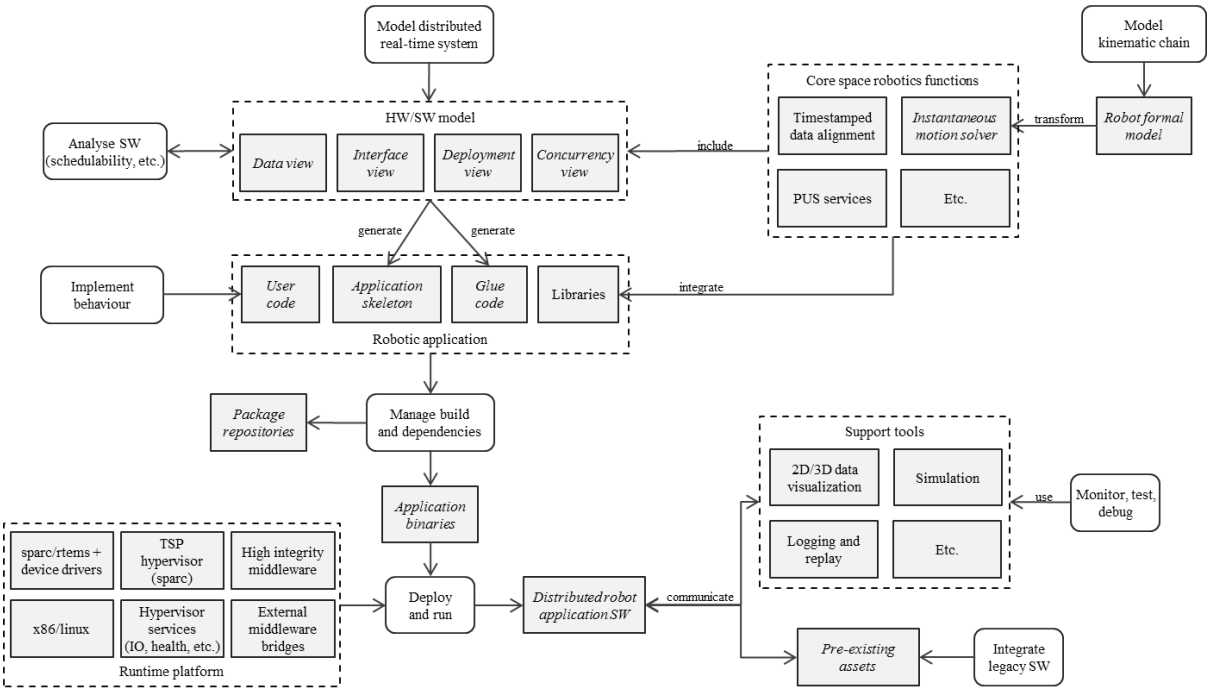


Figure 4-2. Development of a robot control application with ESROCOS

The starting point of the workflow is the formal modelling of the robot and the application. The model-based approach facilitates the early verification of the system properties, in particular for RAMS. The modelling activities encompass the following aspects:

- The robot's kinematic chain, in order to produce a formal model of the robot motion, from which software can be automatically generated.
- The hardware and software architecture of the application, including non-functional properties (real-time behaviour, resource utilisation, etc.).

The models allow for different analyses to verify the non-functional properties of the system and iteratively refine the system architecture. ESROCOS relies on both existing [RD.6] [RD.7] and newly developed tools to support the different modelling aspects.

The model of the application may include functional building blocks, either provided by ESROCOS or specifically generated from the models (e.g. a hybrid dynamics instantaneous motion solver).

This model can then be used to automatically generate the software scaffolding for the application, consisting of the skeleton of the application components and the glue code that enables the inter-component communication. The application-specific behaviour is implemented and integrated in this structure, making use of libraries to support the required functionalities.

The application binaries can then be built and deployed on the desired runtime platform. The application may be distributed, with different components running in separate nodes or partitions. ESROCOS supports SPARC/RTEMS and x86/Linux platforms. The former is intended for usage in space-quality systems, while the latter aims towards laboratory setups as well as validation and debugging purposes.

The framework includes the autoproj package and build management system to handle the builds and the component dependencies. The management system allows the developer to seamlessly combine ESROCOS, 3rd-party and own components to build an application.

ESROCOS can be used to model applications using time and space partitioning, in order to build mixed-criticality systems in which components with different RAMS levels can

safely coexist. These applications can be deployed on a SPARC (LEON) platform using the AIR hypervisor and deployed in space-quality systems.

The communication between the application components at runtime is enabled by the PolyORB-HI middleware, part of the TASTE toolset. Communication can be local or across partitions and nodes.

ESROCOS provides also bridge components that enable the communication between PolyORB-HI and external middleware, in particular for ROS and ROCK systems. This allows the robotics engineer to use tools from these ecosystems (data visualizers, simulators, etc.) for testing and debugging the application. A selection of tools is provided ready to use with ESROCOS, with all the required data types and interfaces. In addition, the middleware bridge allows the user to integrate existing software assets and run them together with newly built software in a distributed environment.

Finally, ESROCOS provides support for importing and exporting software components from the ROS and ROCK frameworks, in order to facilitate the migration of legacy code to the framework.

The components of the ESROCOS framework are defined according to the activities supported by the framework and described in the PDD [AD.6]. This section recapitulates the software components identified in the PDD.

In order to describe the components of the ESROCOS framework, it is necessary to distinguish between two parts or views of the system:

- Robot Control Operating System (RCOS): ESROCOS provides a runtime framework to support the execution of robotics applications, including an operating system, communications middleware and runtime services (or libraries) for common robotics functionalities.
- RCOS Development Environment (RDEV): ESROCOS provides also a set of tools, such as model editors, code generators or data visualizers, to support the development and validation of robotics applications.

Many components of the framework reflect this duality. For instance, the kinematic chains modelling software is a modelling tool (RDEV) that generates code that is integrated in the application (RCOS).

The components of ESROCOS are also classified according to their scope into laboratory and space quality components. Laboratory components are intended for use in non-critical systems and run on a regular Linux workstation. Space quality components are tools or libraries targeted for critical systems, and are developed to a higher level of quality, and are in line with ECSS standards [RD.17].

Finally, the ESROCOS framework combines existing and newly developed components. Depending of the scope of the work foreseen for each component, a different level of detail is provided in the design: new development, extension and integration of existing software, and integration of existing software.

The Table 4-1 enumerates the components of the ESROCOS framework, classified according to these categories.

Table 4-1. ESROCOS software components

Activity	Component	RCOS	RDEV	Lab	Space	Description	Scope of the work
Model kinematic chains	Robot modelling tools	X	X	X	X	Solvers for all possible kinematics and dynamics transformations of lumped parameter robot chains are generated from formal and semantically validable models.	New development
Model and analyse distributed real-time systems	TASTE	X	X	X	X	Framework for model-driven SW development of real-time systems. The main components are: - Orchestrator - ASN.1 compiler - Ocarina - Editors - PolyORB-HI (middleware) - HW library - SDL tools - RTEMS	Extension and integration of existing SW
	BIP compiler		X	X		Compiler tool for generating C++ code from BIP models.	Extension and integration of existing SW
	BIP engine		X	X		Runtime for executing C++ code generated from BIP models.	Extension and integration of existing SW
	TASTE2BIP		X	X		Generation of BIP models from TASTE models.	New development.
	SMC-BIP		X	X		Statistical model-checker for BIP models.	Extension and integration of existing SW
Common robotics functions	Base robotics data types	X		X	X	Elementary data types for robotics applications.	Extension and integration of existing SW
	OpenCV	X		X		Computer vision library.	Integration of existing SW
	Eigen	X		X		Linear algebra library.	Integration of existing SW
	Transformer	X		X	X	Library to support component developers with geometric transformations.	New development
	Stream aligner	X		X	X	Library to support component developers with temporal alignment of time-stamped data streams.	New development
	PUS services	X		X	X	Implementation of the following PUS services in TASTE: - TC verification - Housekeeping TM - Event management - Function management - Time management - Connection test - Timeline-based scheduling - OBCP - Parameter management - File management	New development

Activity	Component	RCOS	RDEV	Lab	Space	Description	Scope of the work
Deploy and run	AIR	X			X	ARINC-653 hypervisor.	Extension and integration of existing SW
	HAIR		X		X	AIR emulator and tools	Extension and integration of existing SW
	CAN bus driver	X			X	Driver for the GR CAN controller for RTEMS	Extension and integration of existing SW
	Ethernet driver	X			X	Driver for the GR Ethernet controller for RTEMS	Extension and integration of existing SW
	SpaceWire driver	X			X	Driver for the GR SpaceWire controller for RTEMS	Extension and integration of existing SW
	EtherCAT driver	X			X	Support for EtherCAT in RTEMS, TASTE, AIR.	New development
Monitor, debug, test	Data logger		X	X		Tool for logging data from TASTE components	New development
	vizkit3d		X	X		3D data visualization	Integration of existing SW
	RVIZ		X	X		Data and image visualization.	Integration of existing SW
	Gazebo		X	X		Robot simulator.	Integration of existing SW
	PUS console		X	X		GUI application to show PUS communication in the control PC.	New development
Integrate legacy SW	Middleware bridges	X	X	X		Tools and libraries to support a runtime bridge between TASTE and the ROS/ROCK environments.	New development
	Framework import tools		X	X		Tools and libraries to support the importing of components from ROS/ROCK frameworks into ESROCOS.	Extension and integration of existing SW, new development
	Framework export tools		X	X		Tools and libraries to support the exporting of components from the ESROCOS framework to ROS/ROCK.	Extension and integration of existing SW, new development
Manage build and dependencies	Autoproj		X	X		Software package management and build tool.	Integration of existing SW
	ESROCOS development scripts		X	X		Collection of development tools for setting up/editing projects in the ESROCOS environment.	Extension and integration of existing SW

The following sections describe the design of each of the high-level components identified in Table 4-1 organized per supported activity, and detailed according to the scope of the planned work.

5. SOFTWARE COMPONENT DESIGN

5.1. MODELLING OF KINEMATIC CHAINS

"Motion" and "perception" are two essential functionalities in any robotic system, and both have a rich and mature history. In other words, there is more than enough understanding about what constitutes a solid basis of theory and algorithms to include in a digital platform for robotics. For the motion part, this basis is given by the generic algorithm to solve the "hybrid" instantaneous kinematics and dynamics relationships that hold between the forces and the motions on ideal, lumped-parameter, kinematic chains. For the perception part, a similar role is played by Bayesian graphical model templates that relate raw sensor data to the task-centred features of objects involved in the robots' tasks.

ESROCOS provides the capability to model a robot's kinematic chain, in order to produce a formal model of the robot motion, from which software can be automatically generated. The generated software are functions that solve a particular query about the robot's kinematics, and that can be integrated in TASTE models.

The Figure 5-1 gives a graphical overview of the underlying modelling approach in the developed toolchain. The first Figure depicts three transformations between models: first, a static model of the kinematic chain must be made; this model is transformed into a model of instantaneous kinematic transformations, from given inputs (in joint space or Cartesian space) to computed outputs (for example, forward position kinematics). A typical use of such functions is motion control, or motion planning: the third transformation adds desired motion profiles (LIN, ARC, PTP) and motion targets, as well as the motion interpolation functions (trapezoidal velocity profile, or "S-curve"); finally, these motion trajectories are executed in a feedback motion control loop, which requires the addition of a control library, an allowed error budget model ("tolerances") in the motion accuracy, and a small selection of other "Quality of Service" specifications (such as distance to singular configurations, or optimality according to a specified redundancy resolution criterion).

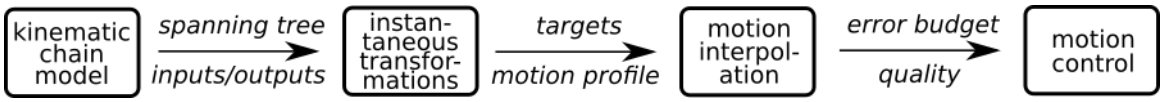


Figure 5-1. Model transformations

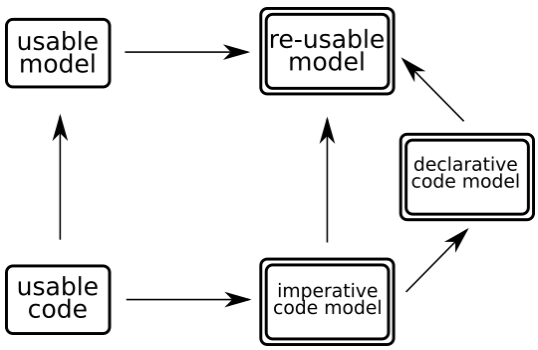


Figure 5-2. Model types and conformance

The Figure 5-2 shows a general schema, representing models/code, that applies in general to any of the steps of the first figure. While in the first figure, the arrows represent

model-to-model transformations (with the annotations representing added model information), in the second figure, the arrows represent "conforms-to" relations: every entity and relation in the "start" node model follows all the constraints represented in the "end" node model. The reusable models have a richer semantics, they account for a variety of cases, whereas the usable ones are specialized for specific use cases. Reusable models can have declarative as well as imperative versions of code; a selected small set of "solvers" will be provided to generate imperative versions of the code from a declarative version. For example, at the level of instantaneous transformations, a usable model might account only for *the* base frame and *the* end-effector frame of a manipulator, whereas a re-usable model allows *any* number of frames *arbitrarily* attached to the kinematic structure, to act as base frame and/or end effector frames, for *any* number of kinematic chain transformations. A declarative code model refers to the pose and velocity of some robot links as a *function* of the joint status and one or more Cartesian landmark positions (e.g., minimal distance to obstacles, or minimum gravity impact on particular links), whereas the imperative model represents the *sequence* of actual operations required to compute such information. The transformation from a declarative specification to an imperative one requires a "solver" that has an (imperative!) algorithm to generate imperative operations from a set of constraints and optimization functions. The transformation from an imperative *model* to actual usable *code* ("grounding") requires additional user configuration, such as the choice of a programming language, and the choice of data types ("digital data representation").

The "kinematic chain" part of the (left-most part of the) first figure contains only *data* (and no operations or relations), so the "code" level has only data structures; the Collada format (or rather, the "kinematic family" part of it) fits well to this part of the tool chain, and a Collada [RD.23] interpreter will be provided.

The "user friendly" part of the instantaneous transformations are the traditional forward and inverse kinematics, for velocity and forces, since these are the only physical entities for which a kinematic chain represents the instantaneous transformations. (The instantaneous transformations contain also *operations*, and not just data.) The case of "Inverse kinematics for position" is covered by the Cartesian motion *interpolation,* because, physically, going from one position to another cannot be done instantaneously, so a motion interpolation *and* control is required; this holds for, both, simulated motions and real motions, the only difference being the constraints on the interpolation and control parameters which determine "stability" and "performance" of the computations. The motion profiles for which implementations will be provided are "LIN" (linear motion in Cartesian space), "PTP" (linear motion in joint space) and "ARC" (circular motion in Cartesian space).

The rest of the section focuses on the tools developed around the "kinematic transformations" part illustrated before. Purpose of the tool is to generate "usable" C code (based on a larger "reusable" code library, that will be provided in full, too) implementing a kinematics solver, starting from a model of the kinematic tree. This code is fully compatible with hard real—time constraints by having a predictable execution time and by avoiding dynamic memory allocation, as well as any kind of IO. The decision to separate the modelling and coding in a "usable" and "reusable" part is to be able to serve the immediate goals of ESROCOS with the "usable" part, shielding its users from the deeper technical details and functionalities of the "reusable" part; the latter is designed to serve the more advanced use cases of other OGs and expected future projects.

A detailed report on the proposed approach and the design for the modelling of kinematics chains is provided in the technical note [RD.37].

5.1.1.DESIGN OF THE MODELLING TOOLCHAIN

The Figure 5-3 gives an overview of the current tool. It consists of "offline" programs, that is executables that will not be directly part of the actual runtime robot control. These programs are Unix command line tools, working with configuration/input text files.

The user-defined inputs are the robot model (kinematic tree model e.g. in Collada format – but more formats might be supported thanks to automatic conversions) and a query. The query would specify the required *kinematic function* (e.g. forward position kinematics of frame X1 and X2, inverse kinematics of tool frame) as well as a set of "grounding choices" required to identify uniquely the working code to be generated.

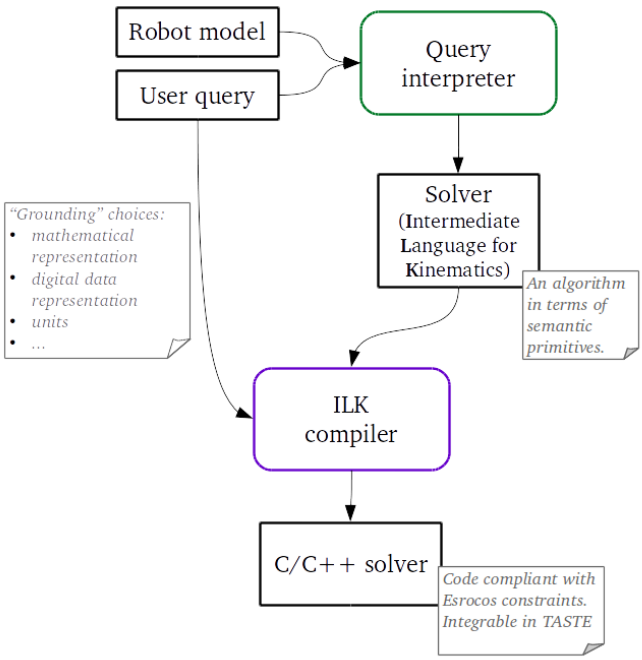


Figure 5-3. Overview of the tool prototype

These grounding choices include the units to be used (e.g. SI units) and the preferred mathematical representation for certain quantities (e.g. quaternion vs rotation matrices to represent relative orientation of frames). The input robot model has to include all the information required to properly interpret the grounding choices, for example by specifying the units being used in the model itself (in the case of numerical constants).

The main function components of the toolchain are the model/query interpreter and the ILK compiler. ILK stands for Intermediate Language for Kinematics. "Intermediate" here refers to somewhere in between the two opposite extremes of a purely symbolic model of a solver (e.g. forward kinematics for "this" arm) and working code for a particular platform. We are designing this modelling language(s) as *the* representation of the "reusable" code library, so full roundtripping is expected for the model-to-code and code-to-model transformations; this roundtripping expectation is based on the hypothesis that we have made the right choices in the decoupling of the solver tool from the grounding choices, that is, the programming language, physical units, digital representations. This decoupling is essential for a toolchain that is *configurable,* that has to generate *correct-by-construction* code, from which the original model can be regained via symbolic meta data in that generated code. With reference to Section 5.1 above, the ILK model is a reusable declarative code model, which our compiler turns into usable code; compilers are indeed the most mature form of declarative-to-imperative transformations, so we

reuse that mature base wherever possible; what we have to add on top of compiler functionalities is the "graph based" reasoning required for the above-mentioned "grounding" of code to meaning, and back.

The interpreter parses the robot model and the query, and generates a model of the desired kinematic solver (cf. the block in the figure labelled "*Solver (ILK)*"); the query interpreter typically exploits prior knowledge about the robot model, in order to perform some optimizations that will eventually result in more efficient code. As not every solver lends itself to such optimizations, the interpreter might be also used solely to check the consistency of the given inputs. In this case, the solver at the ILK level will be developed by hand; this is the case for the standard inverse kinematics for a manipulator end effector.

5.1.2. RUNTIME COMPONENT DESIGN

The final output of the toolchain described in the previous section is plain C/C++ code with predictable execution time, no dynamic memory allocation nor any kind of I/O.

Code will be generated as a set of functions whose exact signature will depend on the original query submitted by the user. The generated solver, for its internal calculations, will use its own data types, plain C numeric types, structures and Eigen data types. Eigen is a C++ library for linear algebra which is part of the set of external libraries chosen for the ESROCOS framework.

The integration of the solver(s) with TASTE will happen as follows: a single TASTE-function with PI (Provided Interface) matching the C functions will wrap the generated code. The PI signature will use the corresponding ASN.1 generated data types which constitute the "standard" within the ESROCOS-TASTE framework. Simple type conversions will be necessary to make the wrapping of the generated solver by such a TASTE function possible.

The PI type will be *unprotected*, as multiple input/output parameters are in general required (e.g. robot joint status as input, end-effector frame as output). There will not be any periodic task (e.g. no internal "behaviour" of this TASTE-function) as the solver is a functional component to be exploited by other parts of the system which may in turn be constrained by specific scheduling (e.g. a joint level controller).

Although this point was not illustrated in the figure before, an additional, simple component of the toolchain will also generate the AADL specification of the TASTE-function wrapping the solver, as well as the underlying glue code (in C) to wire together the PI and the solver interface.

5.2. MODELING AND ANALYSIS OF DISTRIBUTED REAL-TIME SYSTEMS

5.2.1. TASTE IMPROVEMENTS

TASTE is a central component of ESROCOS toolset, in charge of supporting middleware functions through the Ocarina toolset led by ISAE; but also to support modelling activities, and let designers model their system prior to code generation.

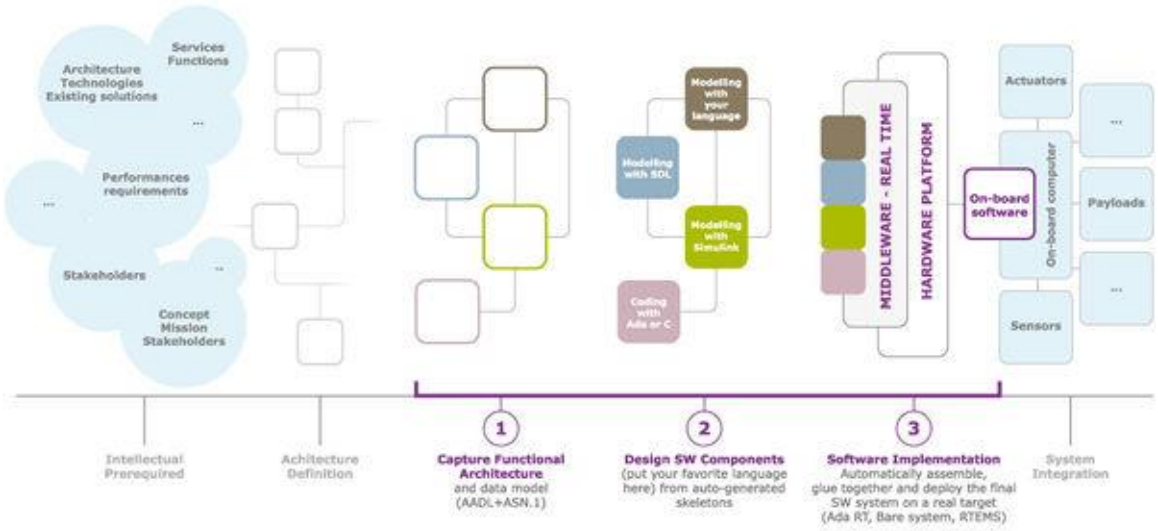


Figure 5-4. The ASSERT process

As defined, TASTE supports the ASSERT process, that aims at automating steps that were perceived as translations from one domain to another. Figure 5-4 recaps the main logic of the ASSERT process. This process is divided up into three phases:

1. A modeling phase, where the developer captures the functional (interfaces, types, etc.) and non- functional properties of the system,

2. A model transformation and verification phase, which verifies the feasibility of the system,

3. An automatic code generation phase which produces a distributed real-time software system that is ready for download on hardware target.

Point 1. is being updated by Ellidiss as part of OG2 activities. Point 2 and 3 are being supported by ISAE, with help and approval from ESA.

As part of ESROCOS activities, TASTE will be silently improved, in the sense that no new visible features will be developed. Instead, emphasis and effort will be directed towards

- Supporting baselined operating systems: RTEMS, AIR;
- Proper integration and testing of drivers done by partners, validation they meet QA criteria depending on Space or Lab quality level;
- Supporting ESROCOS and other OG in their usage of TASTE, and addressing bug reports.

In addition, ISAE is preparing a qualification kit for Ocarina following DO330 guidelines. The set of qualification activities will be aligned with space best practice and requirements for the targeted systems, and will only cover the space-relevant activities. A first set of activities have been performed already and concerned

- The definition of Ocarina tools operational requirements;
- Update of the build procedure to achieve reproducible builds and tests in a controlled environment that meets ESROCOS baseline;

- Addition of static analysis tools to evaluate quality of the code produced, the coverage achieved during test so as to meet expected quality levels for both space and lab settings.

These elements are discussed in deliverable D3.4 Prototyping Report.

As part of the current activities, the architecture of the Ocarina and PolyORB-HI have not been extended significantly, except for two elements reported in this document:

- The PolyORB-HI/C middleware has been extended to support the future RTEMS5 release, and its new driver interface.
- Minor bugs reported by ESROCOS partners have been corrected, full details is published in Ocarina GitHub forge (https://www.github.com/OpenAADL/ocarina).
- TASTE toolset is also being extended to support the integration mechanisms between ROS and TASTE discussed in section 5.6.1.1.

5.2.2.TASTE2BIP

TASTE2BIP is an automatic translation tool of TASTE designs into BIP models [RD.32]. The aim is to validate and verify robotics software designs with formal models and methods, in this case the BIP tools, as illustrated in Figure 5-5. This section is based on [RD.38] which provides a complete description of the translation at theoretical level.

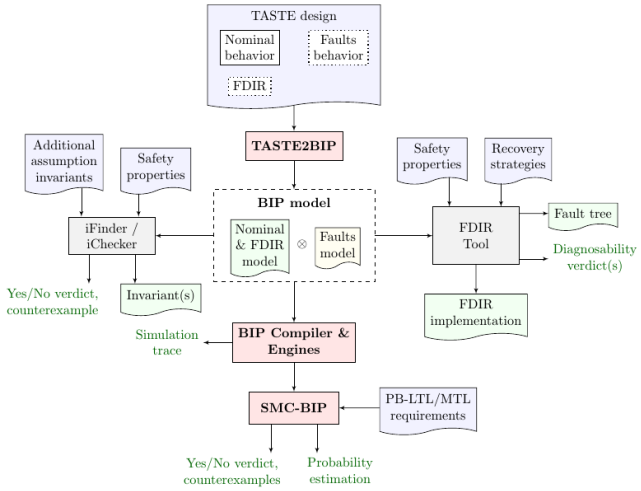


Figure 5-5. The BIP tools. The tools developed within ESROCOS are represented in red rectangles. The inputs of each tool are represented in blue and additionally yellow. The outputs are represented in green.

Figure 5-5 also illustrates a verification and validation workflow with the BIP tools. The main input is a TASTE design which can contain FDIR components. (Such components could be automatically obtained with the FDIR tool under development in OG2). Therefore the analysis can be done on a complete model, provided the fault behavior is specified either directly in TASTE or in BIP. The analysis workflow for TASTE designs with FDIR functionality is described in Section 5.2.5.

The (obtained) BIP model can be subject to compositional verification with the iFinder/iChecker tool. This tool allows to check the satisfaction of safety properties by computing invariants in a compositional manner and interacting with an SMT-solver to check for satisfiability. This tool is currently under development in OG2.

Validation can be done via simulation (stepwise, real-time, interactive, etc.) with the BIP Compiler and Engines. The modifications required by these tools to accommodate real-time and/or stochastic systems are described in Section 5.2.3.

Additionally, the SMC-BIP tool can be used to check qualitative and quantitative probabilistic requirements specified in Probabilistic Bounded Linear Temporal Logic (PB-LTL) and Metric Temporal Logic (MTL). Based on the specified confidence parameters, the tool statistically computes and executes a number of simulations and gives yes/no verdicts or the probability of satisfaction of the requirement, respectively. This tool is redesigned and extended with different features as described in Section 5.2.4.

5.2.2.1. THE TRANSLATED TASTE SUBSET

The input of TASTE2BIP is a TASTE design consisting of data view, interface view and the behaviour of each component in the interface view provided as an SDL state machine or "leaf" C functions. We briefly describe hereafter each of these elements and the modelling constraints they must satisfy.

The data view allows modelling user-defined types in the ASN.1 notation. These types are based on TASTE default types such as (unsigned) integers in 8 and 32 bits, reals and Booleans. ASN.1 allows defining sequences, arrays, enumerations, and strings.

The interface view models the architecture of the system in terms of communicating components (i.e., software functions in TASTE terminology). A component is the basic modelling element describing the functionalities of the system. Components can be atomic or hierarchical. Hierarchical components are defined as the composition of other components.

The behaviour of a component is given as an SDL state machine (i.e., the language attribute must be set to SDL from all the options TASTE allows) or as C code. For this it can define and use parameters of predefined type, user-defined type or Timer.

The functionalities of a component are available to its environment via provided interfaces. A provided interface is in fact an interaction point which allows for receiving signals triggering the functionality associated to that request. It can be of two types:

- cyclic. The associated functionality is executed periodically as defined by the attributes period, deadline and worst-case execution time. A cyclic provided interface cannot define any parameters.
- sporadic. The associated functionality is executed when the call is issued by another component, therefor sporadically. A sporadic interface defines the attributes minimal inter-arrival time (similar in semantics to period for cyclic interfaces), deadline, worst-case execution time and queue size (as requests are queued and handled 1-by-1). This interface can define at most one parameter sent by the other component. In consequence, if several parameters need to be sent one should define a composed ASN.1 data type and use here.
- protected. The associated functionality is executed when the call is issued by another component, and the execution happens in the thread of the caller. The execution can happen only if the caller can get the resource of the protected object. These interfaces can be declared for TASTE components implemented in C (they are forbidden for SDL-based components). Such an interface can have multiple in and out parameters and defines deadline and worst-case execution time attributes.

Requests can be made by other components through required interfaces connected by wires to provided interfaces. The direction is usually from a required interface to a provided one.

For an interface view the following rules must be satisfied. Every component defines at least one provided interface. All sporadic/protected provided interfaces must be connected to a different required interface. In an interface view there should exist at least one cyclic provided interface.

As mentioned above, the functionalities of a component should be given as SDL state machines. A state machine is mainly defined by a set of locations and transitions between locations. A transition in SDL has the following pattern: `trigger; (condition | action)*`. A transition has a run-to-completion execution, only the defined locations being "accessible" to the other functions.

A trigger is either the reception of a signal via a provided interface or a continuous signal. A continuous signal is defined as a predicate over the function's variables and it is executed when the predicate is evaluated to `true`.

A condition is generally a (set of) predicate(s) on the function's parameters. It can have multiple branches and only one can be evaluated to `true` at a moment. That branch is consequently fired during the execution.

The actions to be performed on a transition are *tasks* (i.e., assignments), *sending signals* and *calling procedures*. Indeed an SDL state machine can define parameters (similar to the function parameters including Timers) which can be assigned complex mathematical expressions. Procedures possibly with in/out arguments can be defined in a state machine and they are also designed by state machines. A few predefined procedures exists, such as `write/writeln`, `set_timer` and `reset_timer` for printing messages, setting a value to a timer and resetting to 0 a timer, respectively.

Additionally, components implemented in C are considered for the translation. These components can defined cyclic, sporadic and protected provided interfaces, but it is expected that in their implementation there are no calls to required interfaces. We call such components "leaf" C components.

Finally, TASTE allows defining a deployment view. This view defines how components are mapped to hardware. This view is not considered in the TASTE2BIP translation. However, it can be used for the manual refinement of the application BIP model (the BIP model obtained from TASTE2BIP) into a distributed BIP model. The refinement process will make use of a library of components for obtaining the distributed model. But, as mentioned above, this process is not considered for automation.

Please note that the constraints described above are not imposed by the translation, their aim is to define well-formed TASTE designs in general. Some of the constraints are already enforced by TASTE (e.g., an SDL-based component cannot model protected provided interfaces). Others (e.g., a design must have at least one cyclic provided interface) are defined in order to have *meaningful* input TASTE models. By *meaningful* we mean that these designs are executable and functional, and therefore can be analyzed either by the BIP tools or the TASTE tool-chain, and the analysis results are relevant. TASTE2BIP translates any TASTE design into a possibly partial BIP model.

5.2.2.2. THE BIP LANGUAGE

BIP is a textual formal language allowing to describe timed automata networks [RD.32], the semantics model of a TASTE design.

The definition of a BIP model follows the structure: definition of component types, definition of component interaction types, instantiation of the system – component and interaction -, and definition of priorities. We describe next each of the modeling elements.

A component can be either atomic or compound. An atomic component is described by a state machine i.e., timed automaton. A compound component is described by the instances it contains.

An atomic component can define local typed variables, including clocks. Its computations are represented by a state machine: control locations connected by transitions and on which operations are performed. The locations are enumerated with the keyword *place*.

The maximal time elapse in a control location can be modeled with an invariant. A transition is given by the pattern:

```
on action
  from loci to locj
  [provided (guard)]
  [eager | delayable | lazy]
  [do { (assignment; | conditional; | function_call;)* }]
```

The transition starts by defining the action to be performed, which is either an interaction with other components or a local event. Then we mention the source and target locations of the transition. The transition can be enabled only under some constraints defined by a guard. A guard is a conjunction/disjunction of predicates on data variables and clocks. Next we describe the urgency of the transition: eager means that a transition must be fired as soon as it is enabled, delayable means that the firing of an enabled transition can be delayed up to an upper bound, and lazy means that a transition can be fired at any moment or never. Finally, different instructions can be performed on a transition such as variable/clock assignments with complex expressions, and function calls possibly imbricated in conditionals (e.g., if conditional). The guard, urgency and instructions are not mandatory, but at most one can be present for each.

The actions a component is involved in are defined by ports. A port can define typed parameters involved in interactions and computations. Ports that are involved in interactions are defined as exported, while the others are standard local ports. Ports are also typed: port types are defined in the model and a component instantiates them by providing a name and specifying which of its local data is affected by that port.

Interactions between components are defined as connectors. For a model we can define different connector types corresponding to the interactions needed, which are then instantiated when defining compound component types. A connector can define complex interactions, e.g., whether all components need to synchronize or only a subset and which are the possible subsets. Moreover connectors can define local variables and computations on local and port variables.

Finally, compound component types such as the system are described by the instances of owned ports, components and connectors. Connectors can link the ports of the different components, but they can also involve ports of the compound. Therefore, we can express in BIP hierarchical, open systems.

The order of execution of enabled interactions can be constrained with priorities. One can specify between two given interactions which has the higher priority and in which cases the rule applies. The priority mechanism is internal to a compound.

5.2.2.3. IMPLEMENTATION DETAILS

The implementation of the TASTE to BIP model transformation requires getting access to the various AADL and SDL constructs that define the TASTE input model.

According to the technologies already available inside the current TASTE tool-chain, two main options can be considered for parsing the AADL subset of the TASTE model:

- The Ocarina python binding
- The LMP AADL toolbox used in the IV, DV and CV editors

Ocarina is an AADL compiler and target language code generator that is written in Ada. It implements an AADL parser that generates an internal AST in memory. A python library has been developed to give access to this AST from external applications written in Python. One possible approach is thus to implement the TASTE to BIP model transformation rules in python and use the Ocarina binding to get access to the TASTE entities.

The LMP (Logic Model Processing) technology ([RD.33],[RD.34]) is a generic approach to handle heterogeneous models and meta-models. It consists of representing each meta-model by a list of Prolog predicates and expressing each model processing function by a set of Prolog rules. Figure 5-6 shows the general principles of an LMP connector.

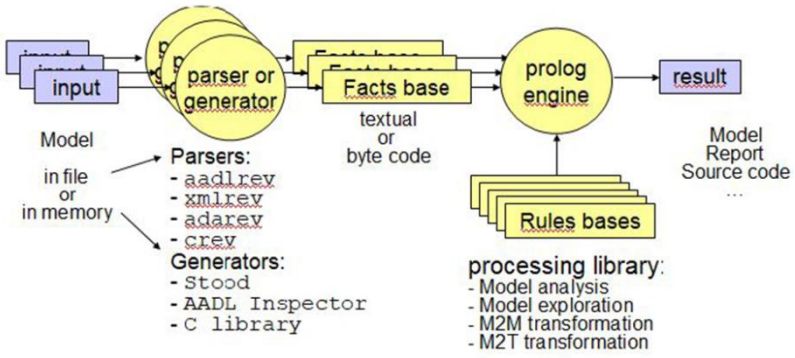


Figure 5-6. The LMP technology

All the components of the LMP toolbox that are required to process AADL models are already included into the TASTE tool-chain. One of the benefits of this approach is that it allows for a declarative and formal implementation of the transformation rules, which makes it appropriate for certification processes. For example, verification tools developed with LMP have been qualified by Airbus according to DO 178 recommendations for the A380 and A350 programs.

In both cases, accessing the SDL subset of the TASTE models will require to reuse directly or indirectly the parser that is implemented in the OpenGeode editor that is part of the TASTE tool-chain.

The python binding in Ocarina being not fully stable in the current version of the TASTE tool-chain, it was decided to use the LMP solution. The LMP implementation of the TASTE to BIP transformation has been split in four separate modules. The Main module only acts as a launcher. The TASTE_AADL module provides access to the TASTE Interface View constructs. The BIP module defines the BIP entities to be produced The SDL2BIP module gives access to the SDL model associated with each TASTE function and generates the corresponding BIP code. The TASTE_AADL and SDL2BIP modules rely on the aadlrev and sdlrev parsers, respectively. The implementation of the translation is done in collaboration with Ellidiss as part of OG2, while the aadl and sdlrev parsers are fully developed by Ellidiss. The code can be obtained at https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/bip/TASTE2BIP.git.

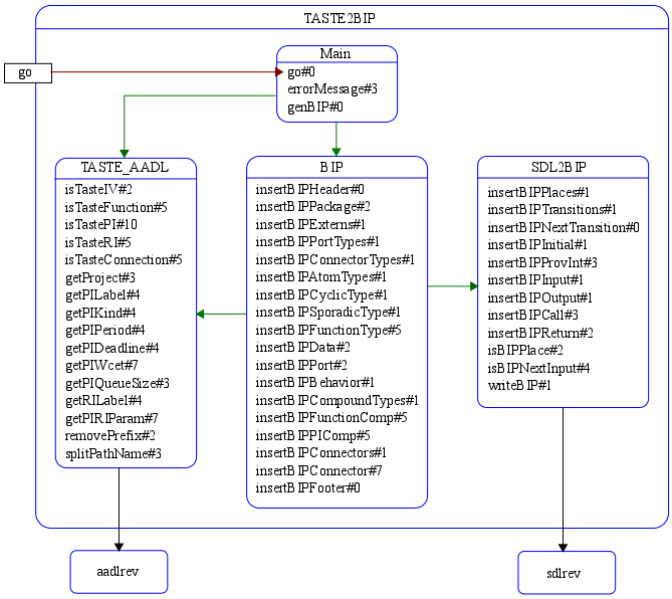


Figure 5-7. TASTE2BIP architectur

In [RD.38] we show how the translation rules are implemented quite straightforwardly in Prolog. All the transformation rules are compiled into a single Prolog byte code file named `TASTE2bip.sbp` that simply needs to be added to the `config/plugins` directory of the TASTE editor installation.

Additionally, to facilitate the access to this new feature from within the editor, a new script called `TASTE2bip.tcl` must be placed inside the `config/externalTools` directory of the TASTE editor installation.

No other action is required. A new Tools menu item is then available to activate the BIP model generation from the currently loaded TASTE Interface View model, as shown in the following screenshot.

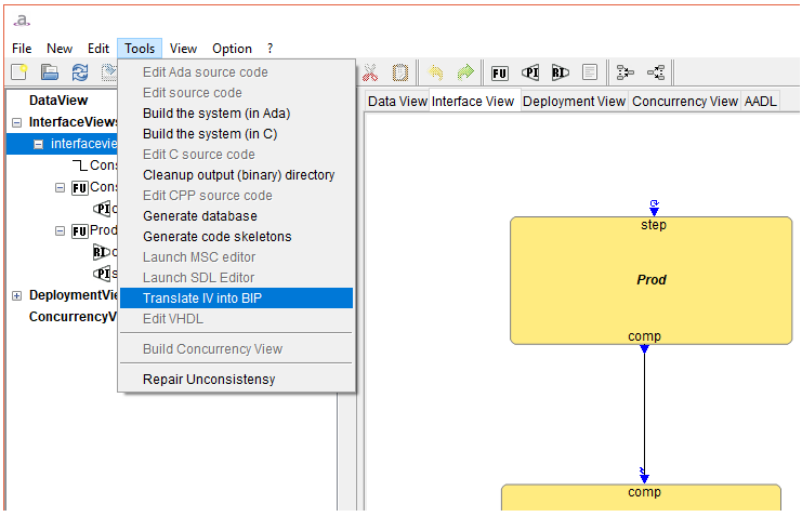


Figure 5-8. Integration of TASTE2BIP in the TASTE editors

5.2.3. BIP COMPILER AND ENGINES

The BIP compiler and engines are the core tools supporting the BIP language. The compiler allows to generate code (in C++ for example) and the engines linked to this code allow to execute it and obtain different simulation types. In the following we detail the extensions required to handle real-time and stochastic systems both at language level and in the execution engines.

5.2.3.1. REAL-TIME COMPILER AND ENGINE

Real-time BIP is a variant of BIP which extends the tools with respect to the time dimension. Such an extension is implemented as a dedicated branch in the Git repository (https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/bip/compiler.git) which is kept as much as possible up-to-date with respect to the master branch. This real-time extension affects most of the modules of the tool-chain (see Figure 5-9).

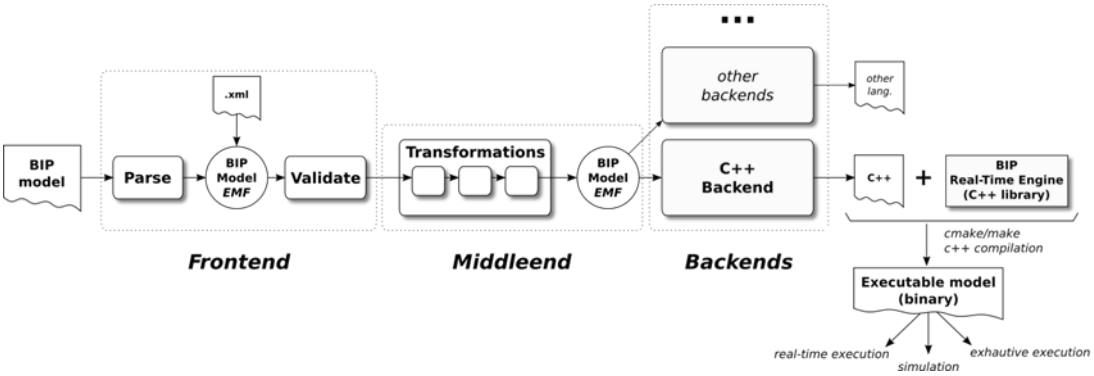


Figure 5-9. BIP Compiler and Engines tool-chain

- In the frontend, the grammar of the BIP language is extended to include clocks in components and timing constraints in automata describing components behaviour. The internal representation of BIP models had also to be modified accordingly.
- The error management is extended to include additional errors specific to the real-time extension of the language.
- In the C++ backend the code generator for atomic components is modified to include the runtime computation of their timing and to provide the result at their interface. These timing constraints are taken into account at runtime by the execution engines to produce correct schedules.
- The execution engines are largely modified and extended compared to their "untimed" equivalent to *(i)* compute global timing constraints of interactions (synchronizations) from timing constraints of atomic components and to *(ii)* ensure correct and global scheduling of models in both simulated time and real-time.

5.2.3.1.1. MODIFICATIONS OF THE META-MODEL

The internal representation of BIP models relies on EMF and is defined by a meta-model. Figure 5-10 gives the main modifications of this meta-model introduced for the real-time extension. The package time is created for time related aspects (clocks, time values, timing constraints, resume constraints, urgencies), whereas the other packages of Figure 5-10 were already present in the untimed version of BIP. Clocks can be declared in atomic components and can be involved in guards of transitions and in state invariants. The difference between *GuardedUntimed* and *Guarded* is in the OCL constraints associated to these abstract classes: the guard in *GuardedUntimed* should not involve clocks, whereas in *Guarded* it can be any arbitrary Boolean expression satisfying the timed automata model (e.g. clocks should not appear in both sides of disjunctions, operator

"not equal" should not be used on clocks, etc.). Of course a guard should also be well-typed. This is controlled by a set of operators and functions allowed by the BIP language, specified directly in Java. Clocks can also be manipulated on transition executions with two operations: modification of their current values and modification of their speed.

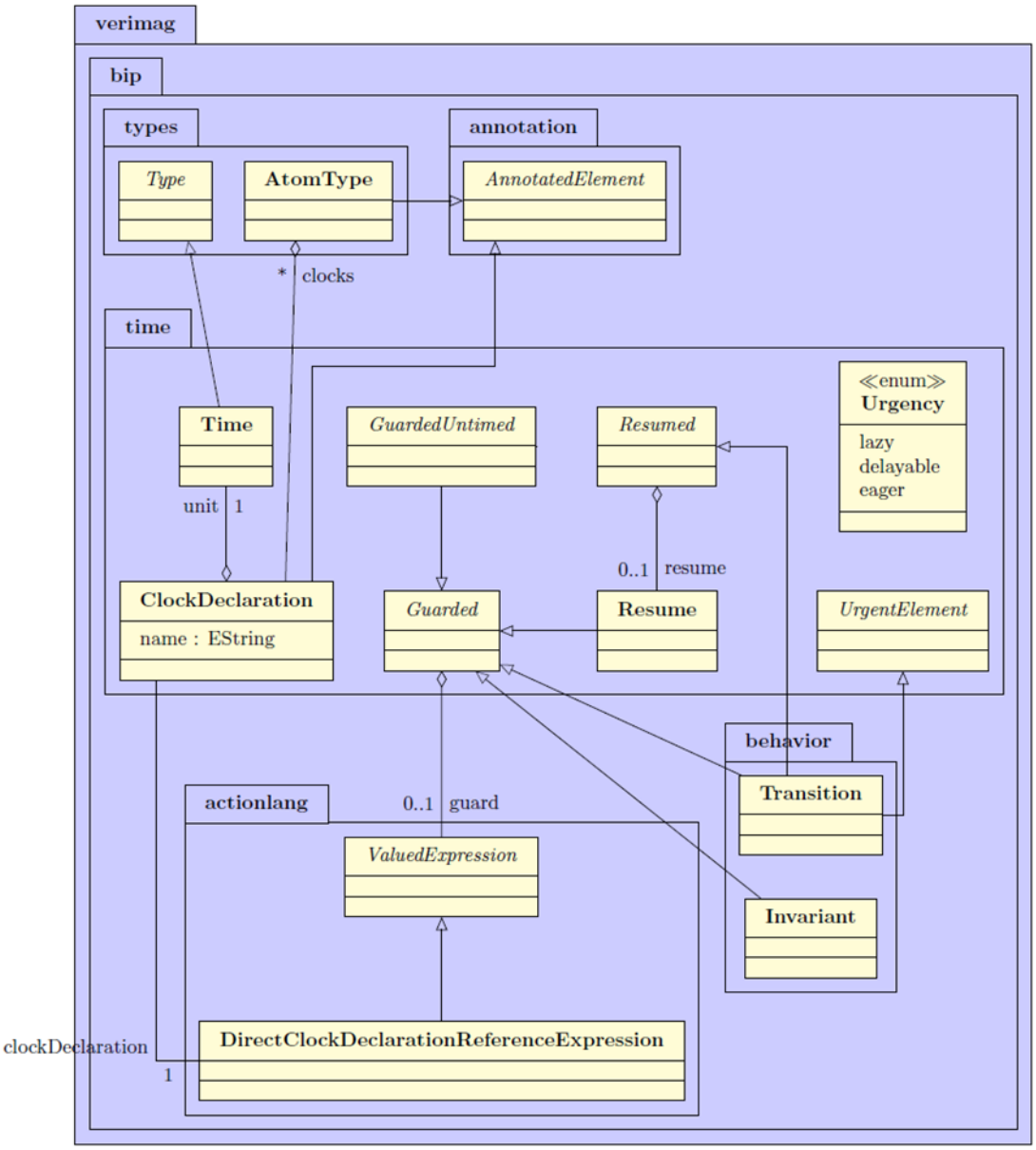


Figure 5-10. Class diagram of the real-time extension of the meta-model

5.2.3.1.2. MODIFICATION OF THE C++ BACKEND

In terms of code generation, the real-time extension affects essentially the generation of the code of atomic components.

- Each clock declared in an atomic component corresponds to an instance of the C++ class *Clock*, whose implementation is provided by execution engines.
- The interfaces of atomic components has been modified to expose timing constraints associated to enabled ports, represented as intervals on global time and an urgency parameter.
- The C++ code generated for atomic components dynamically translates guards involving local clocks into intervals on global time when updating the status of the

interface after a transition execution. It also translates any operation on a clock by the corresponding method call of the class *Clock*.

5.2.3.1.3. MODIFICATION OF THE EXECUTION ENGINES

The C++ code generated from BIP models includes execution mechanisms for atomic components, but only structural information for connectors and hierarchical components. The execution of a model requires additional code to compute, at a given state, the possible interactions (i.e. synchronizations) enabled in the connectors of the hierarchy of components of the model. Interpreters for connectors, priorities and hierarchical components are implemented in separate libraries a.k.a. *execution engines*. Execution engines can be used for several purposes including: simulation, real(-time) execution, debugging, and state-space exploration. The following engines are currently available.

- The reference engine is a safe and straightforward implementation of the semantics of BIP, without taking into consideration performance aspects. It should be considered as a reference for the semantics and its performance is acceptable for executing small models.
- The optimized engine is an optimized version of the reference engine, implementing mechanisms such as caches to avoid the recomputation of the whole interactions/priorities layer after each execution.
- The multithread engine is a multithreaded version of the optimized engine. It relies on the *C++11* standard for implementing threads (whose total number is an input parameter) and uses *atomic* operations for implementing communications and synchronizations. Not only component execution is parallel but also the code responsible for scheduling interactions. Such a design is not the most efficient for centralized platforms, but it would allow targeting distributed platforms by a minimal set of modifications of the code.
- Recently a version of the multithread engine with a centralized scheduler is being developed.

All the above engines have been extended and modified to implement the semantics of the real-time extension of BIP. Figure 5-11 gives a brief overview of the main classes introduced by such an extension, in which most of the attributes and methods have been hidden to avoid overloading the representation. Time values are represented and manipulated through the class *TimeValue*, which in practice offers 64 bits representation with a saturated arithmetic and a precision up to nanoseconds. Timing constraints are represented by combinations of a time interval and an urgency attribute (no urgency by default). At runtime, engines manipulate the current global time in the model through an instance of *ModelClock*. It is updated with respect to the current value of the clock of the platform when necessary. Depending on the execution mode (simulation or real-time), the platform clock implements simulated time or is bound to the real-time clock of the platform. When the platform is too slow for guaranteeing the timing constraints, a time-safety violation exception is generated. The class *Clock* implements local clocks of atomic component used by the generated C++ code generated.

Besides those additional classes, the engines had been modified considerably to implement (real-)time functionalities. This includes computing timing constraints of interactions based the timing constraints of atomic components ports, as well as correctly scheduling the execution in order to meet the timing constraints. This is particularly challenging in multithreaded execution where scheduling decisions are made from partial knowledge of the global state. To address this problem we introduced new optimizations in the tool-chain, refining the notion of resources in the multithread engines to distinguish when the state of an individual port or data of component is ready.

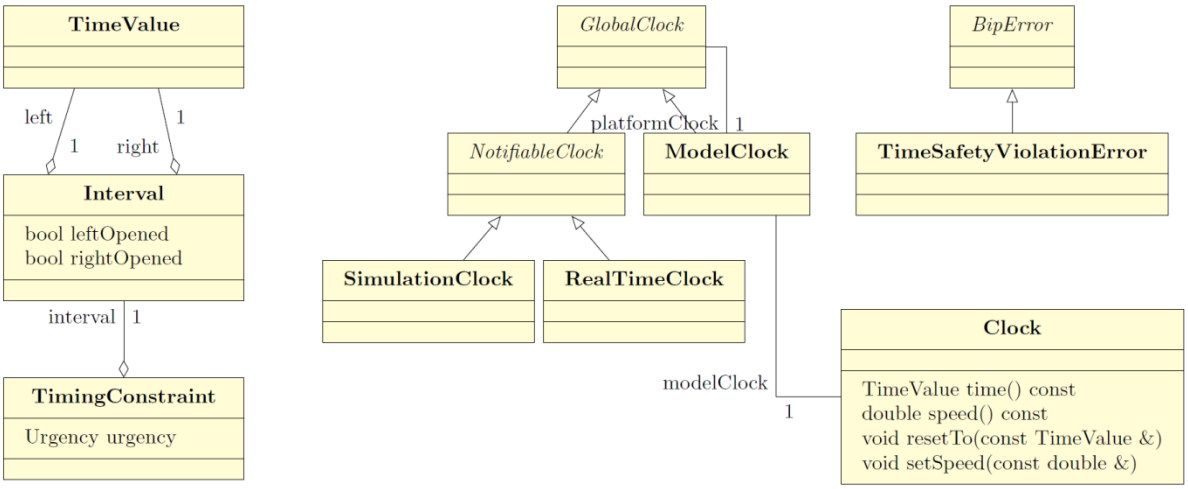


Figure 5-11. Additional classes introduced for the real-time engines

5.2.3.2. STOCHASTIC SIMULATION ENGINE

5.2.3.2.1. OVERVIEW OF THE SYNTAX AND SEMANTICS

The *stochastic real-time BIP* formalism reconciles two distinct extensions of BIP that have been developed in the past. On the one hand, *real-time BIP* extended BIP with real-time features (clocks, urgencies), had dense real-time semantics based on timed automata with urgencies and is used for modelling, analysis and implementation of real-time systems. On the other hand, *stochastic BIP* extended BIP with stochastic features (probabilistic variables), has discrete-time stochastic semantics based on Markov chains and is used mainly for analysis using statistical model-checking methods.

The stochastic real-time BIP allows for defining components as timed automata extended with stochastic constraints. Such components are composed, under specific restrictions, using two categories of interactions, namely *timed* or *stochastic*. Timed interactions are associated only with (pure) timing constraints expressed as lower and upper bounds over clocks valuations, as in timed automata. These interactions are scheduled for execution with respect to an implicit uniform or exponential probability distribution as it is generally the case in several existing frameworks. Stochastic interactions are associated with one stochastic constraint (that is, a user-provided arbitrary density function), defining their occurrence time relative to a clock value. The underlying semantics of stochastic real-time BIP is defined as a Generalized Semi-Markov Process. It produces timed traces $\omega=(a_0,t_0)\ (a_1,t_1)\ \ldots$, where a_i are interactions and t_i are timestamps. A complete formal definition is available in [RD.31].

As an example, Figure 5-12 provides a graphical illustration of stochastic real-time components. On the left, the Receiver component is essentially a timed automaton. On the right, the Channel component contains in addition a stochastic port labelled *rcv_ack* defined by a normal density function, i.e., its scheduling time is sampled with respect to a normal with mean 10 and standard deviation 2.

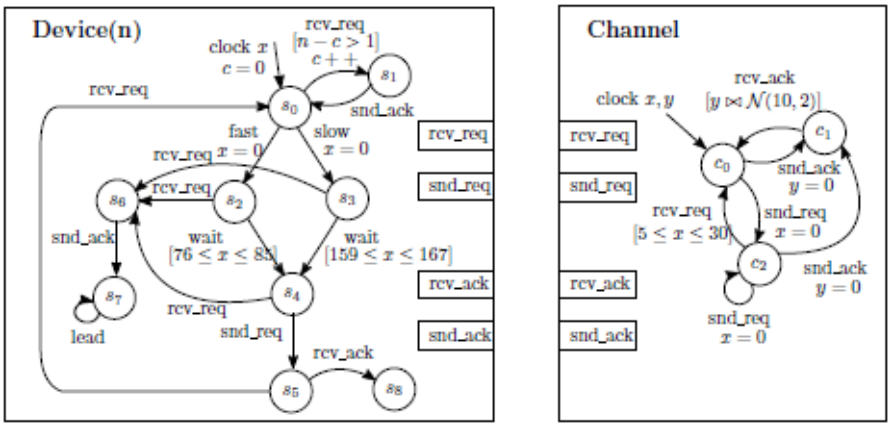


Figure 5-12. Stochastic real-time BIP: Components example

From the language point of view, stochastic interactions are defined using specific annotations `@stochastic(dist="...", clk=...,param="...")` to tag components ports. Such annotations specify a probability density function and its parameters through, respectively, the *dist* and *param* attributes, respectively, and to associate a clock through the clk attribute. For example, the stochastic port *rcv_ack* of the Channel component above is defined by *dist="normal"*, *clk=y*, *param="10,2"*. Currently, the language supports a number of built-in density functions (normal, gamma, and X^2). Additional (empirical) functions, can be used through the same mechanism (*dist="custom"* and in *param* a file characterizing the underlying cumulative distribution).

Below, we model the Channel component with stochastic real-time BIP:

```
atom type Channel(int id)
  /* Data declaration */
  data int id_channel = id
  /* clocks declaration */
  clock x unit nanosecond
  clock y unit nanosecond
  /* ports declaration */
  export port ePort snd_ack( id_channel)
  export port Port snd_req()
  export port ePort rcv_req( id_channel)
  @stochastic(dist="normal",clk=x, param="10,2")
  export port Port rcv_ack()
  /* control locations */
  place c0, c1, c2
  /* transitions descriptions */
  initial to c0
  on snd_req from c0 to c2 do{x=0;}
  on snd_ack from c0 to c1 do{y=0;}
  on rcv_ack from c1 to c0
  on snd_req from c2 to c2 do{x=0;}
  on snd_ack from c2 to c1 do{y=0;}
  on rcv_req from c2 to c0 provided(x<=30 && x>=5)
end
```

5.2.3.2.2. MODULE ARCHITECTURE

The stochastic simulation engine implements the operational semantics of stochastic real-time BIP systems. Given a model *S*, the engine produces system traces consistent with the timing and stochastic constraints in *S*. Traces are generated in two modes, namely, symbol-wise, or at-once for an a priori given length. The first is for online monitoring,

since trace generation can be interrupted as soon as a verdict is obtained. The second is interesting in the context of SMC loop as it bounds the time for obtaining a local verdict.

The functioning of the stochastic simulation engine is depicted in Figure 5-13. At every step, the engine computes the firing (time) interval for every interaction, based on current clock valuations and interaction guards (Evaluate). Next, an execution date is chosen for every future enabled interaction (Plan). For *timed interactions*, the date is chosen by sampling a value in the associated firing interval, using either an uniform or exponential law, depending if the firing interval is bounded or not. For *stochastic interactions*, the date is chosen according to their associated probability density function and the clock value. Two cases are distinguished: when the current value of the clock is zero, the date is chosen by a direct sampling of the corresponding density. Otherwise, when the clock has a strict positive value, the execution date is planned using the truncated density function at that value. Once all the future enabled interactions are planned, the scheduler implements a race policy and selects for execution the one having the earliest planned date. The simulation time is advanced to that date and the interaction is executed on the system (and logged).

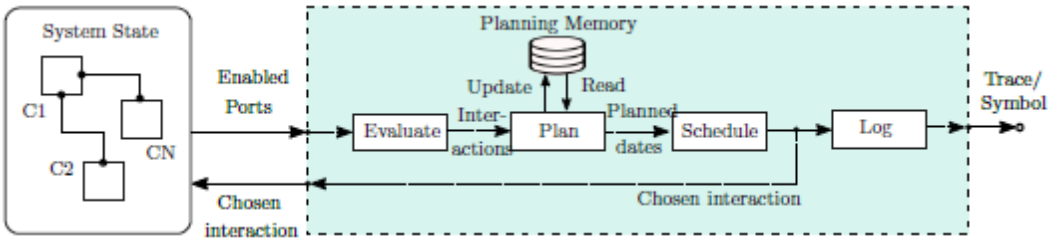


Figure 5-13. Functional view of the stochastic simulation engine

For efficiency reasons, planned execution dates are stored in order to avoid re-planning interactions that remain enabled when moving to the next system state. A new execution date is chosen only for newly enabled interactions and/or in conflict with the executed interaction. That is, when the associated clock (for stochastic interactions) has been reset and/or the firing interval has changed due to execution of previous interaction.

5.2.4.SMC-BIP

In contrast to standard formal verification techniques such as model-checking, Statistical Model-Checking (SMC) techniques are appropriate to evaluate performance metrics (and not only) of a system. SMC can be seen as an improvement of purely simulation-based techniques since it guarantees results with respect to user-defined confidence level. This is obtained in a SMC tool by computing a sufficient number of executions to get the coverage of the system behaviour required by the confidence level. The SMC-BIP tool applies these techniques for an input stochastic real-time BIP model. In the following we detail the tool, its features and new design.

5.2.4.1. OVERVIEW

The new version of SMC-BIP was completely re-designed as an IDE including all the activities from the modelling, to the simulation and the SMC analysis. It offers a set of functionalities organized in a clear and fluid workflow as illustrated in Figure 5-14. All interactions with SMC-BIP go through a graphical user interface (GUI), which allows for setting the inputs, running analysis and getting the outputs.

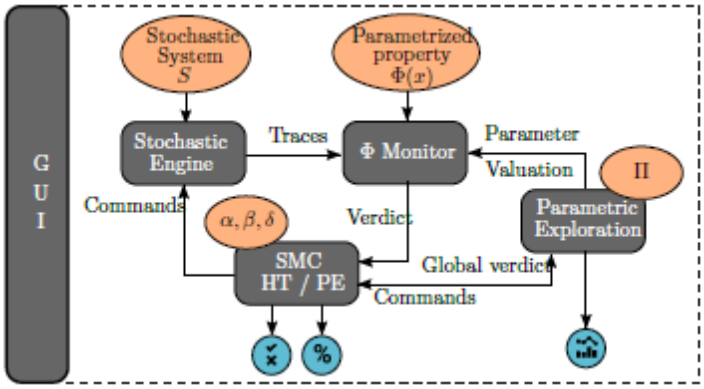


Figure 5-14. SMC-BIP architecture (workflow)

5.2.4.1.1. MODULAR AND EXTENSIBLE ARCHITECTURE

The tool architecture was designed modularly for more flexibility and to enable extensibility. The tool relies on four main generic functional modules, namely, a Stochastic Simulation Engine (see Section 5.2.3.1), a *Monitoring* module, an *SMC Engine*, a *Parametric Exploration* module plus additional data structures, i.e., to represent execution traces and logical formulas. The stochastic engine encapsulates an executable model simulator and is used to produce (random) execution traces on demand. The monitor is used to evaluate properties on traces. The SMC engine implements the main statistical model-checking loop depending on the statistical method used, namely, hypothesis testing or probability estimation. Finally, the parametric exploration module coordinates the evaluation of a parametric property. All these modules are fully independent and interact through well-defined Java interfaces.

The tool has been instantiated for the BIP formalism as input model, where different stochastic simulation engines can be used (discrete/real-time). Regarding monitoring, the tool currently supports bounded LTL and parametric MTL. The functionality of these modules is detailed below.

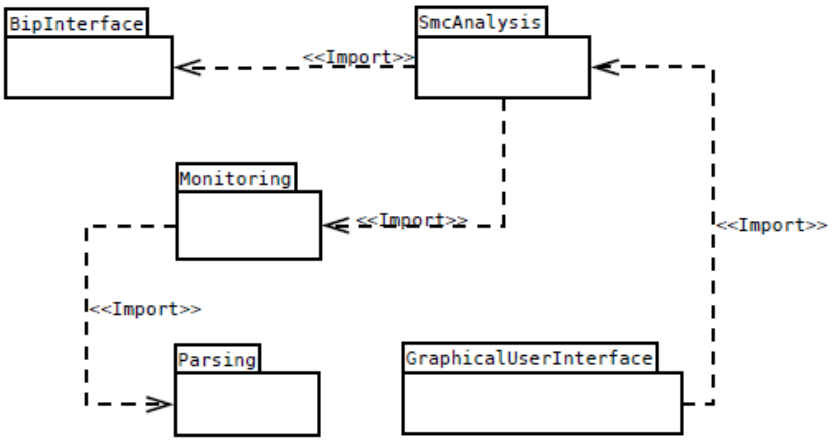


Figure 5-15. SMC-BIP package diagram

From a code organization perspective, the tool is structured in different packages as shown in the package diagram in Figure 5-15. These packages reflect the architecture depicted above. The package *BipInterface* encapsulates the BIP engine and exposes a clean interface to correctly interact with it. The *Parsing* package provides means for parsing input properties (requirements) in the supported temporal logics. The *Monitoring* package groups the classes related to the monitoring activity, e.g. MTL monitoring. The SMC analysis algorithms are organized within the *SmcAnalysis* package. Finally, the *GraphicalUserInterface* package implements all the graphical interface of the tool.

5.2.4.1.2. MULTIPLE INTEGRATED ANALYSIS WORKFLOWS

The tool takes as inputs a real-time stochastic system model to be analysed/simulated, a property of interest, and a set of parameters mainly required by the SMC algorithms. Two analysis workflows are provided by this new version:

- The first one is the classical SMC procedure consisting of either a *Hypothesis Testing* (HT) or a *Probability Estimation* (PE). It consists of triggering the stochastic simulation engine to produce a new execution trace which is monitored against the given property. This produces a local verdict, i.e., regarding that specific execution trace. Depending on the used SMC algorithm, several iterations are generally required to produce a global verdict.
- The second workflow consists of analysing different instances of a parametric property by performing several iterations of the usual SMC workflow.

Details about the two workflows can be found in Section 5.2.4.3 and Section 5.2.4.4, respectively. Depending on the used workflow, one can visualize the analysis results as a single probability, a yes/no answer, or a chart/bar plot. The tool also allows for visualizing the generated execution traces. Storing execution traces can be enabled/disabled by the user as it may be memory consuming. Further details on the features provided by the tool are provided in Section 5.2.4.5.

5.2.4.1.3. TECHNICAL INFO AND AVAILABILITY

SMC-BIP is fully developed in the Java programming language, and requires the Java Runtime Environment (JRE) 7. It uses ANTLR 4.7 (http://www.antlr.org/) for PB-LTL/MTL properties parsing, and the GNU Scientific Library (GSL) 2.3 library (https://www.gnu.org/software/gsll) for probability density functions manipulation. At this stage, SMC-BIP only runs on the GNU/Linux operating systems as it relies on BIP simulation engines. The tool is freely available for download at http://www-verimag.imag.fr/Statistical-Model-Checking.html (the sources are available at https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/bip/sbip2.git). A video tutorial explaining the tool usage can be also found at https://www.youtube.com/watch?v=MvNfZrvlVAs.

5.2.4.2. MONITORING MODULE

5.2.4.2.1. PARAMETRIC MTL SYNTAX AND SEMANTICS

Metric Temporal Logic (MTL) is an expressive temporal logic that extends Linear Temporal Logic (LTL) by introducing an explicit representation of time. MTL temporal operators are similar to LTL with the difference of having a time interval $I \subseteq \mathbb{N}^+$ constraining the temporal operators. For given Ψ, the set of (atomic) state formulas, the syntax of an MTL formula φ is inductively defined by the following grammar:

$$\varphi ::= t \mid f \mid \psi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathrm{o}\varphi \mid \varphi_1 U_I \varphi_2 \mid \varphi_1 R_I \varphi_2$$

where $\psi \in \Psi$.

The operator $\mathrm{o}\varphi$ is the *next* operator, while $\varphi_1 U_I \varphi_2$ is the *until* operator, which stands for φ_1 holds until φ_2 does at any time in I. The *release* operator R_I is the dual of U_I. The *eventually* and the *globally* operators are expressed respectively as $\diamond_I \varphi \equiv t\, U_I \varphi$ and $\Box_I \varphi \equiv f\, R_I\, \varphi$. Their meaning is respectively φ eventually holds at some time in I, and φ always holds at any time in I, respectively.

For the sake of usability, we allow for parametric MTL formula $\varphi(x)$ where x is an integer parameter taking values in some bounded domain Π. The parameter can appear either in state formula Ψ or as bound for time intervals I and is statically assigned a value from its domain before analysis. For instance, $\varphi(x) \equiv \diamond_{[0,t]} [(node3.status = leader)]]$ states that $node3$

eventually becomes the leader before *t* time units, where *t* is the parameter of the property.

5.2.4.2.2. MONITORING MODULE ARCHITECTURE

This module implements the generic infrastructure for online/offline monitoring of properties. At abstract level, the module takes as inputs a formula and either an entire trace or an online stream of trace symbols, and computes a verdict stating whether the trace satisfies the formula. Traces, formulas and symbols are designed as Java interfaces that can be extended with specific implementations.

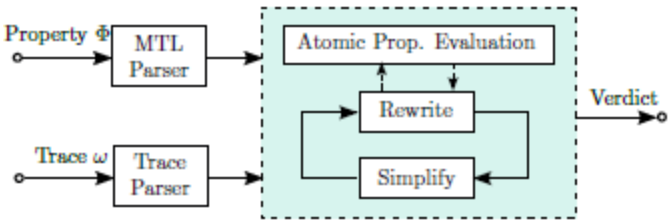


Figure 5-16. Functional view of the MTL Monitor.

In the current version of SMC-BIP, we integrate the monitoring of Bounded LTL (ongoing work) and MTL formulas, untimed and timed BIP traces. Bounded LTL was already included in the first version of the tool. However, it was restricted to formulas without nested temporal operators. At contrary, the monitoring of MTL formulas represents a completely new development. The MTL monitor, illustrated in Figure 5-16, implements an online monitoring algorithm based on the rewriting rules from [RD.25]. Given an MTL formula φ and a timed trace ω, the monitor alternates rewriting and simplification phases. Rewriting consumes a timed symbol $\sigma_i = (a_i, t_i)$ of ω and partially evaluates the current formula φ into φ'. Partial evaluation includes the unfolding of temporal operators and evaluation of atomic state formulas to their truth value. Simplification applies reduction rules on the formula φ' based on Boolean logic (e.g., $t \wedge \varphi' \equiv \varphi'$) so as to conclude or to simplify it as much as possible before the next cycle.

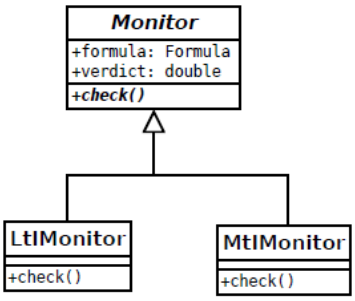


Figure 5-17. Monitor class diagram.

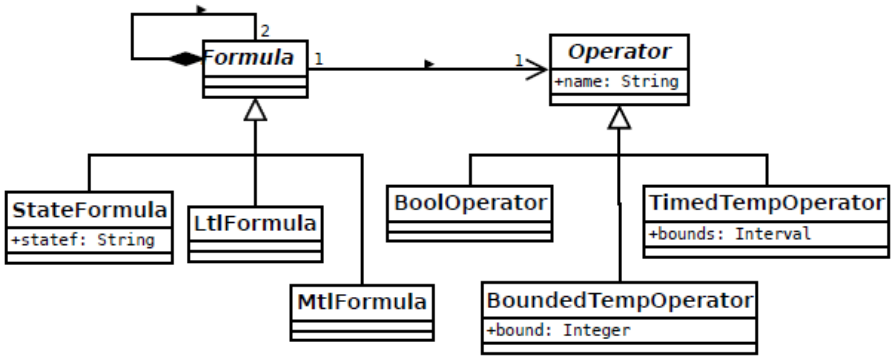


Figure 5-18. Formula class diagram.

5.2.4.3. STATISTICAL MODEL-CHECKING ENGINE

The SMC engine implements several statistical testing algorithms for stochastic systems verification, namely, Single Sampling Plan (*SSP*), Simple Probability Ratio Test (*SPRT*) ([RD.26],[RD.27]), and Probability Estimation (*PESTIM*) [RD.28]. We briefly recall the main procedures to decide whether a given real-time stochastic system *S* satisfies a BLTL/MTL property φ. SMC refers to a series of simulation-based techniques to answer two types of questions: Qualitative - is the probability for *S* to satisfy φ greater or equal to a certain threshold θ? and Quantitative - what is the probability for *S* to satisfy φ?

The main proposed approach to answer the qualitative question is based on *hypothesis testing* [RD.27]. Let p be the probability of $S \vDash \varphi$. To determine whether $p \geq \theta$, we can test $H{:}\, p \geq \theta$ against $K{:}\, p < \theta$. A test-based solution does not guarantee a correct result but it is possible to bound the probability of making an error. The *strength* (α, β) of a test is determined by two parameters, α and β, such that the probability of accepting *K* (resp., *H*) when *H* (resp., *K*) holds is less or equal to α (resp., β). Since it is impossible to ensure a low probability for both types of errors simultaneously, a solution is to use an *indifference region* $[p_1, p_0]$ (with θ in $[p_1, p_0]$) and to test $H_0{:}\, p \geq p_0$ against $H_1{:}\, p \leq p_1$. Several hypothesis testing algorithms exist in the literature. [RD.27] proposes a logarithmic-based algorithm that given p_0, p_1, α and β implements the *Sequential Ratio Testing Procedure* (*SPRT*) (see [RD.26] for details). When one has to test $\theta \leq 1$ or $\theta \geq 0$, it is however better to use *Single Sampling Plan* (*SSP*) (see [RD.29],[RD.27] for details), an algorithm in which the number of simulations is pre-computed in advance. In general, this number is higher than the one needed by *SPRT*, but is known to be optimal for the above-mentioned values. More details about hypothesis testing algorithms and a comparison between *SSP* and *SPRT* can be found in [RD.29].

We also implement the estimation procedure (*PESTIM*) proposed in [RD.28]. It enables to compute the probability p for *S* to satisfy φ. Given a *precision* δ, this procedure computes a value for p' such that $|p' - p| \leq \delta$ with *confidence* $1 - \alpha$. The procedure is based on the *Chernoff-Hoeffding bound* [RD.30].


```
Data: system S,  property φ, confidence params α, δ
Result: A probability estimation pe

Monitor m; Engine e;
nbPos = 0; i = 0;
all = getNbRequiredTraces(α, δ);
while i<N do
  tr = e.generate(S);
  nbPos = m.check(φ ,tr);
  i++;
end
pe = nbPos/N;
```

Figure 5-19. Probability Estimation (PESTIM)

5.2.4.4. PARAMETRIC EXPLORATION MODULE

Parametric exploration is an automatic way to perform statistical model checking on a family of properties that differ by the value of a constant. The family of properties is specified in a compact way as a parametric property $\varphi(x)$, where x is an integer parameter ranging over a finite instantiation domain Π. Figure 5-20 illustrates the different phases of a parametric exploration workflow. The algorithm returns a set of SMC verdicts corresponding to the verification of the instances of $\varphi(x)$ with respect to $x \in \Pi$. This workflow is very useful when exploring unknown system parameters such as, buffers sizes guaranteeing no overflow, or the amount of consumed energy. It automates the exploration for large parameters domains as opposed to tedious and time consuming manual procedures.

```
Data: system S,  parametric property φ(x), instantiation domain Π
Result: A set of SMC verdicts V
Monitor m; Engine e;
V = ∅;
foreach v ∈ Π do
  smc.init();
  while !smc.conclude() do
    tr = e.generate(S);
    verdict = m.check(φ(x), tr);
    smc.add(tr, verdict);
  end
  V = V ∪ smc.getVerdict();
end
```

Figure 5-20. Parametric exploration

5.2.4.5. GRAPHICAL USER INTERFACE

We implemented a user-friendly graphical interface (GUI) that centralizes all the interactions with the tool. The GUI is organized in three main regions: (1) a project explorer, (2) a toolbar and (3) a central panel as illustrated in Figure 5-21.

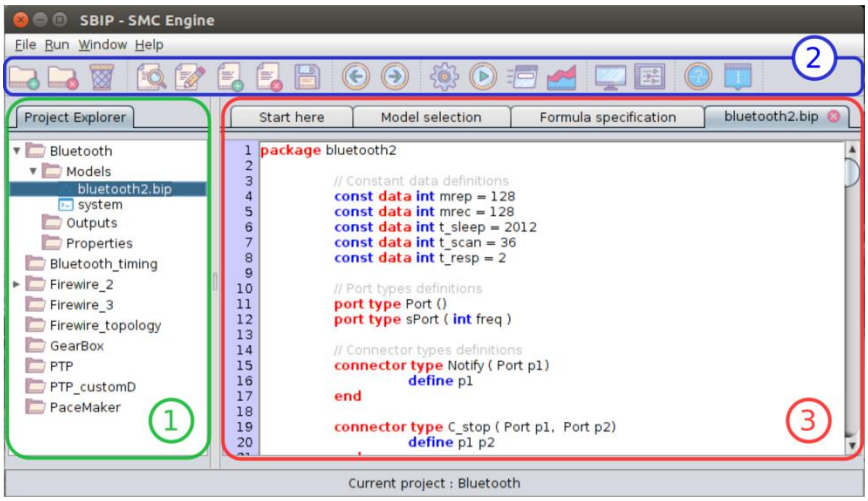


Figure 5-21. Screenshot of SMC-BIP GUI

The project explorer gives a centralized and organized view of the different items of a project during the modelling and the analysis. Such items usually include different files organized in a tree hierarchy. The *Models* folder contains (.bip) models, auto-generated and external (.cpp/hpp) source code, custom probability distributions, and executables. The *Properties* folder stores (.mtl) and (.ltl) properties. Finally, the *Outputs* folder contains execution traces.

The toolbar is organized in six functional groups: (i) project management, allowing to create/remove projects, (ii) file management for model files creation, deletion, edition and visualization, (iii) tab navigation, (iv) workflow management for model compilation, simulation and analysis, (v) configuration setup, and (vi) help buttons.

The central panel is the main region where the designer can load/visualize/edit inputs, configure parameters, run analyses, and visualize results. Each of these operations is provided through a specific view displayed in a separate tab:

- edition view: used to edit models and properties. This also allows for loading and saving various files. Specific capabilities, such as code auto-completion and keyword highlighting, are provided for BIP models.
- configuration view: used to select the simulation engine, the SMC algorithm and parameters, and the instantiation domain for parametric properties.
- analysis view: used to initiate and track the progress of analysis.
- results view: used to provide a summary of the performed analysis, the verdict and/or the set of verdicts on different traces, specific curves and/or plots, overall and partial running times, etc.

5.2.5.FDIR IMPLEMENTATION AND ANALYSIS

This section presents a methodology to use the BIP tools for FDIR Implementation and Analysis, and not a software component. The methodology is illustrated in and briefly discussed in Section 5.2.2.

The process starts with a TASTE design input. In all cases the TASTE design describes the nominal behaviour of the system. Additionally the model can define an FDIR component and the faulty behaviour of the components concerned by FDIR.

The model is translated with TASTE2BIP into a BIP model. The obtained BIP model consists of several packages: one package containing the nominal behaviour, one package containing the faulty behaviour, and one package to describe the requirement to verify.

The nominal behaviour package defines the architecture of the system and the entire behaviour. Please note that the FDIR component resides in this package. If this component defines a behaviour it is translated, otherwise a component with only its interface and empty state machine is defined. Then the user will have to provide such a component in BIP language with eventually external C++ code.

This definition for FDIR components accommodates the use of automatically generated FDIR implementations, such as it is proposed in ERGO. Consider the example provided in Figure 5-22. The TASTE design consist of the nominal behaviour and an FDIR component. The connections between the nominal behaviour and the FDIR are provided beforehand. The FDIR implementation is obtained through some method (manually or automatically) in BIP. Then the BIP model is embedded in the TASTE component, with a wrapper describing the connections between the TASTE function environment and the BIP component environment (e.g., the connection between the provided interface I_p and the BIP port p).

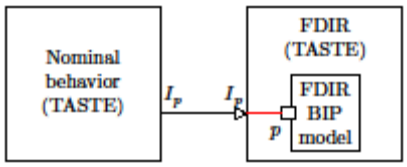


Figure 5-22. Example of integrating generated FDIR implementation in TASTE

The faulty behaviour package defines the faults the system can experience and the behaviour after a fault. This package is automatically generated by the translator empty: it replicates the system architecture for each component (type) including interfaces (ports) and data variables, while the state machine of the behaviour is empty. For the later, the locations from the nominal behaviour's state machine are copied. This helps the user to define the faulty model.

The faulty model could be explicitly designed in TASTE. To identify the faults which are unobservable events, the name of these signals must start with the keyword "fault". Then the faulty model will be translated in a standard way, and will be part of the nominal behaviour package – which becomes an extended behaviour package.

Finally, a requirement package is generated. This package is initially empty. The user can model here the requirements to check as BIP automata, if these cannot be directly expressed in PB-LTL/MTL logic.

Once all the packages are fully defined by the user, two validation tools can be applied on the extended BIP model: SMC, described in Section 5.2.3.1, for statistical model-checking, and iFinder for safety requirement verification. iFinder takes as input the extended model on which it computes an over-approximation of the system behaviour with invariants. Then, the tool checks whether these invariants satisfy the given safety requirement (in Z3 format). A yes/no answer is provided, and in case the requirement is not satisfied a counter-example is produced. If the counter-example is spurious, i.e., the computed invariants are not strong enough, the user can add additional assumptions – invariants in Z3 format – and restart the verification process, until a satisfying answer is provided.

5.3. COMMON ROBOTICS FUNCTIONS

5.3.1.BASE ROBOTICS DATA TYPES

The base robotics data types permit the exchange of relevant information within the framework. Those types have a triple purpose:

- To model the data and interfaces of the robotics application in TASTE, using the Data and Interface Views.
- To define the API interfaces to exchange data between framework components, application components and tools.
- And to serialize and send data through the communication layer provided by the PolyORB-HI middleware.

A set of robotics data types related to geometry, sensor data, actuator control, navigation and guidance have been chosen for implementation in ESROCOS. These types come from the ROCK base-types package and are originally defined in C++, and made accessible in ESROCOS by translating them in ASN.1 and providing a set of conversion functions. The TASTE infrastructure then generates the necessary serialization functions for data transport.

The ASN.1 data types provided by ESROCOS are organized in independent packages according to their scope. Three type packages are provided by the framework:

- **types/base:** basic data types describing robot geometry and motion, imported from the ROCK base-types package.
- **types/sensor_samples:** data types to represent different types of sensor data commonly used in robotics, also imported from the ROCK base-types package.
- **types/drivers:** data types associated to certain devices used in robotics applications (e.g., joystick) and used within the ESROCOS software for testing purposes.

Users can add new data type packages according to the needs of the application, i.e., to model the data and interfaces of additional software components, sensors or actuators. ESROCOS includes tools that support importing data types from ROS and ROCK (see section 5.6.2).

The collaborative development environment of ESROCOS allows users to combine generic functions with their own developed components to build applications. In the case of the data types, a set of CMake macros have been defined to support the creation and usage of new user packages. These macros are located in the ESROCOS **buildconfig** package (inside *esrocos.cmake*).

- **`esrocos_asn1_types_package`** `(<name> [[ASN1] <file.asn> ...] [OUTDIR <dir>] [IMPORT <pkg> ...])`

  CMake function to build an ASN.1 types package in ESROCOS.

- **`esrocos_asn1_types_build_test`** `(<name>)`

  CMake function to create an executable for the encoder/decoder unit tests generated by the ASN.1 compiler.

- **`esrocos_asn1_types_install`** `(<name> [<prefix>])`

  CMake function to install the ASN.1 type files into the ESROCOS install directory.

- **`esrocos_pkgconfig_dependency`** `(<target> [<pkgconfig_dep_1> ...])`

  CMake function to create a target dependency on a library to be located with pkg-config. It tries to find the library, and applies the library's include and link options to the target.

The Table 5-1 summarizes the types provided by ESROCOS in the **types/base** package. In addition to common robotics data types, this package includes a *TASTE-extended.asn* file that extend the default *TASTE-types.asn* definitions with integer, floating point and string types of different sizes.

Table 5-1. Basic robotics data types (types/base)

File	Data Type	Data Description	Definition/Fields
Angle.asn	Angle	Angle in radians	rad: T-Double
	AngleSegment	Angle segment in radians	startRad: T-Double width: T-Double
BodyState.asn	BodyState	State of a rigid body state [Pose] in SE(3) with uncertainty information	ref-time: Time pose: TransformWithCovariance velocity: TwistWithCovariance
Covariance.asn	Covariance	Covariance matrics	Matrix6d
Eigen.asn	Affine3d	Affine transform (double)	4x4 matrix
	AngleAxisd	Rotation expressed as axis direction (vector) and angle in radians	Vector of length 4
	Isometry3d	Isometric transform (double)	4x4 matrix
	Matrix2d	2x2 matrix (double)	
	Matrix3d	3x3 matrix (double)	
	Matrix4d	4x4 matrix (double)	
	Matrix6d	6x6 matrix (double)	
	MatrixXd	NxN matrix (double)	Max. size is 20x20
	Quaterniond	Quaternion (double)	
	Transform3d	Transform matrix (double)	4x4 matrix
	Vector2d	Vector of length 2 (double)	
	Vector3d	Vector of length 3 (double)	
	Vector4d	Vector of length 4 (double)	
	Vector6d	Vector of length 6 (double)	
	VectorXd	Vector of length N (double)	Max. length is 100
JointLimitRange.asn	JointLimitRange	Specification of Joint limits for a robotic model position	min, max: JointState
	OutOfBounds	Error information for out of bounds joint	error-name: T-String min, max, value: T-Double
JoinLimits.asn	JointLimits	Static size vector of JointLimitRange	vector-name: T-String ranges: [JointLimitRange]
Joints.asn	Joints	Static size vector of JointState for multiple optionally named joints.	timestamp: Time names: [T-String] elements: [JointState]
JointState.asn	JointState	Joint State structure with position [double], speed[position/s], effort [N or Nm], raw (i.e. PWM signal) and acceleration[rad/s^2 or m/s^2]	position: T-Double speed, effort, raw, acceleration: T-Float
JointsTrajectory.asn	JointTrajectory	Sequence of JointState	[JointState]

File	Data Type	Data Description	Definition/Fields
	JointsTrajectory	Structure to hold a time-series in the form of vector of JointState, for multiple optionally named joints.	vector-name: T-String timetags: [Time] trajectory: [JointTrajectory]
	InvalidTimeStep	Time step error	time-step: T-UInt32
Motion2D.asn	Motion2D	Motion command for ground vehicles moving in a 2.5D space	translation, rotation: T-Double
NamedVector.asn	InvalidName	Name error	nameString: T-String
Point.asn	Point	3D point (meters)	Vector3d
Pose.asn	Position	3D position (meters)	Vector3d
	Position2D	2D position (meters)	Vector2d
	Orientation	3D orientation	Quaterniond
	PoseUpdateThreshold	Update threshold for pose	distance, angle: T-Double
	Pose	Position (Point [meters]) and orientation (Quaterniond) of a robot pose	pos: Position orient: Orientation
	Pose2D	Position (Point2D [meters]) and orientation (rad) of a robot pose	position: Position2D orientation: T-Double
Pressure.asn	Pressure	Pressure sample (Pascals)	timestamp: Time pascals: T-Double
RigidBody Acceleration.asn	RigidBodyAcceleration	RigidBodyState with acceleration information	ref-time: Time acceleration: Vector3d cov-acceleration: Matrix3d
RigidBodyState.asn	RigidBodyState	State of a rigid body state [Pose] in Affine3d with uncertainty information per each element of the Pose (position and orientation)	timestamp: Time sourceFrame, targetFrame: T-String pos: Position orient: Orientation velocity: Vector3d angular-velocity: Vector3d cov-orientation, cov-velocity, cov-angular-velocity: Matrix3d
Temperature.asn	Temperature	Static size temperature stored in Kelvin (SI unit)	kelvin: T-Double
Time.asn	Time	Time stamp codified as single int64_t number representing microseconds from 1970-01-01T00:00:00	microseconds: T-Int64 usecPerSec: T-Int32 (constant)
Trajectory.asn	Trajectory	Trajectory defined in a Spline form.	speed: T-Double points: [Point]
TransformWith Covariance.asn	TransformWithCovariance	Transform3D with uncertainty	translation: Position orientation: Quaterniond cov: Covariance

File	Data Type	Data Description	Definition/Fields
TwistWith Covariance.asn	TwistWithCovariance	Twist (Vector3d linear velocity [m/s] and Vector3d angular velocity [rad/s]) with uncertainty information	vel, rot: Vector3d cov: Covariance
Waypoint.asn	Waypoint	Points representation for a Pose in a path	position: Vector3d heading, tol-position, tol-heading: T-Double
Wrench.asn	Wrench	Vector3d force [Newtons] and Vector3d torque [Newton meter]	timestamp: Time force, torque: Vector3d
Wrenches.asn	Wrenches	Named vector of Wrench	vector-name: T-String timestamp: Time wrenches: [Wrench]
TASTE-extended.asn	T-Double	REAL type matching range of C double	
	T-Float	REAL type matching range of C float	
	T-Int16	INTEGER type matching range of C int16_t	
	T-Int64	INTEGER type matching range of C int64_t	
	T-String	OCTET STRING of bound size	Max. size is 256
	T-StringN	OCTET STRING of parametrizable size	
	T-UInt16	INTEGER type matching range of C uint16_t	
	T-UInt64	INTEGER type matching range of C uint64_t	
TASTE-types.asn (provided by TASTE)	T-Boolean	Boolean type	
	T-Int32	INTEGER type matching range of C int32_t	
	T-Int8	INTEGER type matching range of C int8_t	
	T-UInt32	INTEGER type matching range of C uint32_t	
	T-UInt8	INTEGER type matching range of C tint8_t	

The Table 5-2 summarizes the types provided by ESROCOS in the **types/sensor_samples** package.

Table 5-2. Basic sensor data types (types/sensor_samples)

File	Data Type	Data Description	Definition/Fields
CompressedFrame.asn	CompressedFrame	Timestamped image in compressed format	ref-time, received-time: Time image: [T-UInt8] attributes: [Frame-attrib-t] datasize: Frame-size-t frame-mode : Frame-compressed-mode-t (enumerated)

File	Data Type	Data Description	Definition/Fields
			frame-status : Frame-status-t
DepthMap.asn	DepthMap	DepthMap image from sensory data (e.g. LIDARs)	ref-time: Time timestamsp: [Time] vertical-projection, horizontal-projection: PROJECTION-TYPE (enumerated) vertical-interval, horizontal-interval: [T-Double] vertical-size, horizontal-size: T-UInt32 distances, remissions: [T-Float]
DistanceImage.asn	DistanceImage	Regular grid depth map image	ref-time: Time width, height: T-UInt16 scale-x, scale-y, center-x, center-y: T-Float data: [T-Float]
Frame.asn	Frame	Data structure representing a visible camera image and its metadata [pixels].	frame-time, received-time: Time image: [OCTET STRING] attributes: [Frame-attrib-t] datasize: Frame-size-t data-depth, pixel-size, row-size: T-UInt32 frame-mode: enumerated frame-status: enumerated
	FramePair	Stereo image pair	
	Frame-attrib-t	Key-value attribute [string]	data, att-name: T-String
	Frame-size-t	Frame size [pixels]	width, height: T-UInt16
IMUSensors.asn	IMUSensors	Information from an Inertial Measurement Unit(IMU) acceleration in [m/s], gyroscopes [rad/s] and magnetometers [Tesla]	timestamp: Time acc: Vector3d gyro: Vector3d mag: Vector3d
LaserScan.asn	LaserScan	Laser scans measurements from a laser sensor	ref-time: Time start-angle, angular-resolution, speed: T-Double ranges: [T-Int32] minRange, maxRange: T-UInt32 remission: T-Float
Pointcloud.asn	Pointcloud	Static size vector of Points	ref-time: Time points: [Point] colors: [Vector4d]
Sonar.asn	Sonar	Representation of data acquired by a Sonar sensor	ref-time: Time timestamps: [Time] bin-duration: [Time] beamWidth, beamHeight: Angle bearings: [Angle] speed-of-sound: T-Float bin-count, beam-count: T-UInt32 bins: [T-Float]
SonarBeam.asn	SonarBeam	Representation of data acquired by one beam of a Sonar sensor	ref-time: Time bearing: Angle sampling-interval: T-Double speed-of-sound: T-Float beamwidth-horizontal, beamwidth-vertical: T-Float beam: [T-UInt8]
SonarScan.asn	SonarScan	Representation of data acquired by a Sonar sensor	ref-time: Time data: [T-UInt8] time-beams: [Time] number-of-beams, number-of-bins: [T-UInt16]

File	Data Type	Data Description	Definition/Fields
			start-bearing, angular-resolution, beamwidth-horizontal, beamwidth-vertical: Angle speed-of-sound: T-Float memory-layout-column, polar-coordinates: T-Boolean

The Table 5-2 lists the types provided by ESROCOS in the **types/drivers** package. These types are likely to evolve and be structured in different packages later in the project.

Table 5-3. Basic driver data types (types/drivers)

File	Data Type	Data Description	Definition/Fields
CanOpen.asn	CanMsg-T	CAN message data (fixed size)	OCTET STRING
	Can-MsgType-T	Message type configuration for the CANOpen protocol	
	Can-BaseId-T	Base ID configuration for the CANOpen protocol	
	Can-Defs-T	Configuration of message definitions for the CANOpen protocol.	
JoystickCommand.asn	JoystickString	Fixed size string type for joystick labels	OCTET STRING
	AxesVector	List of joystick axes names and values	names: [JoystickString] elements: [T-Double]
	ButtonVector	List of joystick button names and values	names: [JoystickString] elements: [T-UInt8]
	JoystickCommand	Joystick input data: device, time, axes and buttons	deficeIdentifier: T-String base-time: Time axes: AxesVector buttons: ButtonVector
pohicdriver-can	Can-Conf-T	CAN driver configuration for PolyORB-HI-C	
pohicdriver-ip	IP-Conf-T	TCP driver configuration for PolyORB-HI-C	
TimestampEstimatorStatus.asn	TimestampEstimatorStatus	Status of an estimator	

In order to use these types in TASTE models, they must be imported into the model's Data View using the TASTE-update-dataview command.

A set of utility functions is provided for the types. In order to use them, component developers must have access to the base types. From a toolchain perspective, the types and support functions are placed in a library that enables loading either from a library package or from code generated by TASTE.

Utility functions are provided for a subset of the types in the **types/base** and **types/sensor_samples** packages. These functions are located in **types/base_support** and **types/sensor_samples_support**, which are libraries that can be imported and linked into any ESROCOS application.

Two types of functions are provided:

- Conversion functions: convert between the ROCK C++ types and the equivalent ESROCOS C types generated from ASN.1.
- Utility functions: functions to operate with the ESROCOS C data types.

Conversion and utility functions have been added to the support packages according to the needs of the implementation. For instance, type conversion functions are provided for those types that are used in the interfaces with ROCK components such as vizkit3d. Additional support functions may be added later.

The Table 5-4 indicates the ASN.1 types for which support functions are provided:

Table 5-4. Basic robotics data types (types/base)

File	Data Type	Conversion	Utility
base/Angle.asn	Angle	X	
	AngleSegment	X	
base/BodyState.asn	BodyState	X	
base/Eigen.asn	Matrix3d	X	X
	Matrix4d	X	X
	Matrix6d	X	X
	Quaterniond	X	X
	Vector3d	X	X
	Vector4d	X	X
	Vector6d	X	X
	VectorXd	X	X
base/Joints.asn	Joints	X	
base/JointState.asn	JointState	X	
base/Motion2D.asn	Motion2D	X	
base/Pose.asn	Orientation	X	X
	Pose	X	
base/RigidBodyState.asn	RigidBodyState	X	
base/Temperature.asn	Temperature	X	
base/Time.asn	Time	X	
base/TransformWith Covariance.asn	TransformWithCovariance	X	
base/TwistWith Covariance.asn	TwistWithCovariance	X	
base/Waypoint.asn	Waypoint	X	
base/TASTE-extended.asn	T-String	X	X
	T-StringN	X	X
sensor_samples/DepthMap.asn	DepthMap	X	
sensor_samples/Frame.asn	Frame	X	
	Frame-attrib-t	X	
sensor_samples/IMUSensors.asn	IMUSensors	X	
sensor_samples/LaserScan.asn	LaserScan	X	
sensor_samples/Pointcloud.asn	Pointcloud	X	
sensor_samples/SonarBeam.asn	SonarBeam	X	X

5.3.2.OPENCV

OpenCV is a widely-used, open-source computer vision library. It is provided by the ESROCOS framework for use by laboratory applications.

The OpenCV library is provided as-is by the framework. In order to include the library in a user application, the developer should include the library as a dependency, and the build system will include it in the build.

Refer to the documentation of OpenCV [RD.18] for more information: http://opencv.org/

5.3.3.EIGEN

Eigen is a widely-used, open-source C++ library for linear algebra. It is provided by the ESROCOS framework for use by laboratory applications.

The Eigen library is provided as-is by the framework. In order to include the library in a user application, the developer should include the library as a dependency, and the build system will include it in the build.

Refer to the documentation of Eigen [RD.19] for more information: http://eigen.tuxfamily.org/

5.3.4.TRANSFORMER LIBRARY

Geometric transformations and reference frame conversions are ubiquitous operations in robotics systems. To ease these operations, ESROCOS provides a dedicated Transformer library derived from the ROCK ecosystem. There is some overlapping with the kinematics modelling tool described in section 5.1, but they are often used with different purposes. The Transformer tool is primarily intended to operate on the geometric data exchanged by components, e.g. for adapting the reference frames, while the kinematics modelling tool is best suited to model and generate code for queries within a specific component.

The transformer [RD.21] library provides the functionality of representing a graph structure formed by the following elements: A set of vertices V, representing coordinate frames of reference, and a set of edges E, representing transformations between two of those frames. By finding a path between two arbitrary frames $X1,X2 \in V$ it is possible to form a chain of arithmetic frame transformation operations thus calculating the transformation of coordinates given in the coordinate system of X1 into the coordinate system of X2. If more than one path between two frames can be found within the graph, strategies to deal with inconsistencies and to decide the path to base the calculation on will additionally be required. Otherwise, a tree-like structure can be enforced for the graph to guarantee unambiguity as exemplified in .

The transformations defined by the edges of the graph can either be static, meaning that the value of the transformation will not be subject to change over the runtime of the system or dynamic and thus can be updated during runtime and might be invalid at a given point of time if not updated accordingly.

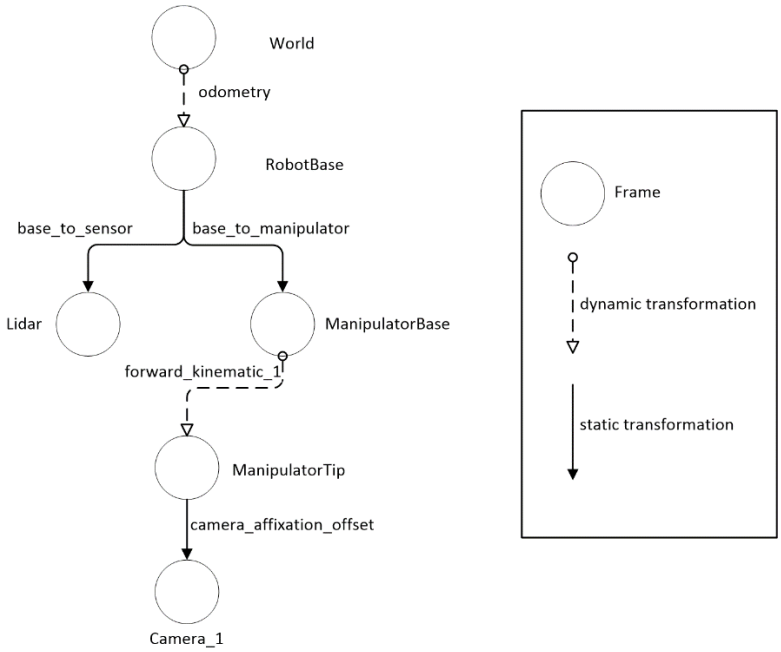


Figure 5-23. An example robotic transformation tree

5.3.4.1. TRANSFORMER LIBRARY

The transformer library will provide the possibility to define such a transformation graph at compile or configuration time as well as values for any static transformations. At runtime, the core functionality is to provide means to update the dynamic transformations and let ESROCOS components query for transformations between arbitrary frames. While data values for the dynamic transforms change during runtime, the overall topology of the transformation graph is considered to remain static. To meet the requirements of ESROCOS the transformer library is designed to avoid dynamic memory allocation, which is planned to be realized by defining the structure of the graph at compile or configuration time, thus allowing for verification of the memory consumption. The transformer library will be built to be statically linked into an ESROCOS system and provide C/C++ interfaces to work with.

5.3.4.2. TASTE INTEGRATION

For integrating the Transformer with TASTE, we plan wrapping as a TASTE component on the same integration level as other components. Configuration can be done by providing ASN.1 data structures for modelling the transformation tree and using them as context parameters. Figure 5-24 shows the design how external interfaces are planned to be modelled in TASTE. Thereby each PI on the Transformer function is mapped the corresponding API call of the library to update a specific dynamic transformation within the transformation graph. For each transform, that was queried from the transformation graph (at design time), a RI is created, which will be called as soon the particular transform could be constructed. The incoming transforms are expected to carry valid time stamps such that the involved data streams can be aligned using the stream aligner.

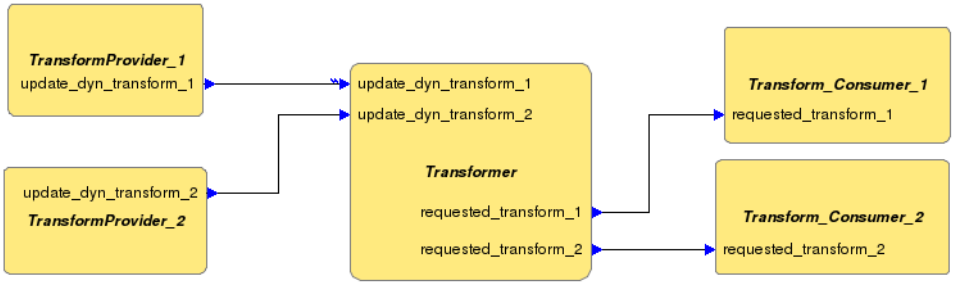


Figure 5-24. Transformer library wrapped by TASTE function

5.3.5.STREAM ALIGNER LIBRARY

Another common issue in the robotic domain is the handling of data that is created and processed asynchronously [RD.20]. To support the handling of such data, ESROCOS implements a mechanism with multiple asynchronous data streams or sporadic Provided Interfaces (PI). Conceptually, alignment of data streams divides in two parts, the estimation of time and alignment of samples. Both concepts are related but separately to each other and their design is described in the following.

5.3.5.1. TIMESTAMP ESTIMATION

In a multisensory data system as a robot, multiple sensors might produce information at different frequency. A typical example is shown in Figure 5-25, where three types of sensors, IMU, LiDAR and cameras are running simultaneously with different periods (10ms, 25ms and 17ms). Timestamping data that originates from sensors is a task that is hindered by phenomenon in the data acquisition chain: sensor acquisition process, communication between the sensor and the CPU, operating system scheduling (when the driver process gets executed once the data arrived) and – last but not least – clock synchronization in multi-CPU (and therefore multi-robot) systems. Few of these can be estimated offline and/or online.

The correct estimation of time is separated in two parts. Inaccuracies produced by the latency (constant part) and inaccuracies from the jitter (variable part). The goal of the timestamp estimator is to reduce the variable part to zero by correcting the time with some information.

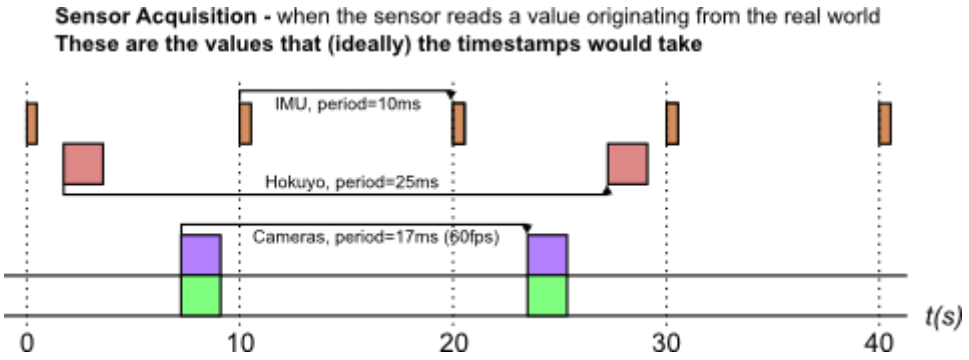


Figure 5-25. Sensor acquisition timeline with three sensors at different frequencies

The timestamp estimator is a submodule or class of the stream aligner, which filters the jitter based on some initial guess. This guess is given by some property values. A graphical interpretation of latency and jitter in data streams is given in Figure 5-26.

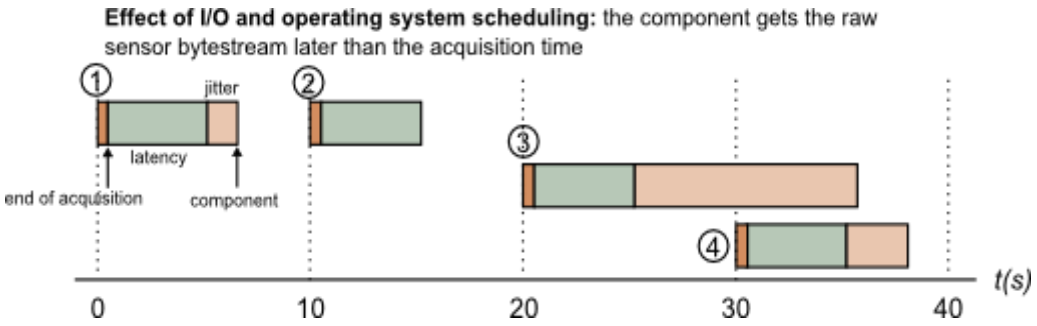


Figure 5-26. Effect of latency and jitter on sensor acquisition time

The average latency, however, is a different issue altogether. It originates from different sources, each of which has to be solved by different means:

Sensor the latency in the sensor acquisition process is usually documented (or can at least be informally given by the company producing this sensor). In case of own-manufactured sensors, this value can be estimated or measured.

Communication layer a rough estimate can usually be obtained by looking at the amount of data and the communication layer bandwidth.

Operating system is the weak part of the chain if one does not use a hard-real-time operating system such as Xenomai, QNX or RTEMS or has devices that are not compatible with that real-time operating system. However, some communication layers timestamp messages at the driver level (for instance, CAN and firewire stacks on Linux systems). For other layers, no information can be obtained directly.

Clock synchronization. Clock synchronization solutions like NTP are available to synchronize multiple CPUs. However, they take long to converge, especially over wireless networks (if they converge at all), making it practical only on systems that are up most of the time. Additionally to NTP, outdoors, one can use GPS as a time source. Indoors, no really good solution exists to our knowledge.

Hardware triggering. It became common for sensors used in robotic applications to have so-called hardware synchronization signals that announce a particular event (for instance, start of acquisition), or hardware triggers that allow to pick the point of acquisition (common on cameras). Using adapted hardware and combined with the techniques proposed above; this method allows achieving data timestamping of the order of one milliseconds, regardless of the operating system properties. The TimestampEstimator class also accepts a separate stream of timestamps that it is using to estimate the latency. Check out the next page for details.

The TimeStampEstimator module is used to remove the jitter out of a periodic time stream. Once configured, one gives it a time in a time series that (1) is periodic (2) can (rarely) contain lost samples and/or bursts, and returns the best estimate for the provided time. One design criteria for this class is that it has to be zero-latency, i.e. the timestamp estimator never delays the processing of a sample.

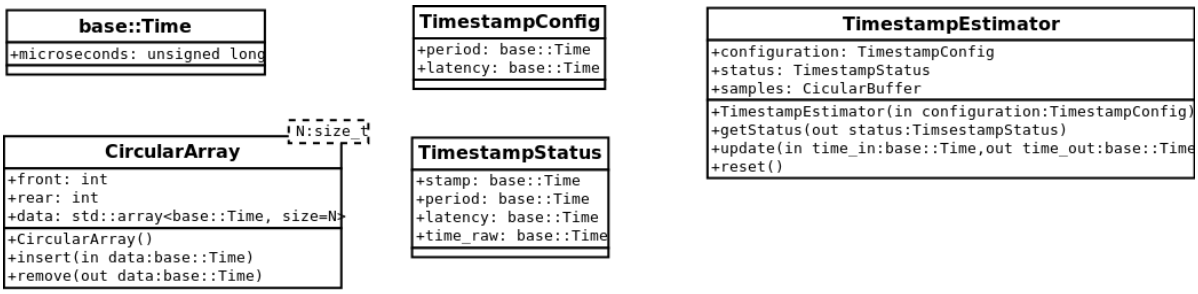


Figure 5-27. Reduced view of the API of the TimestampEstimator

A UML diagram of the module is given in Figure 5-27. The following individual classes or structures can be identified in the diagram:

- base::Time: This is the standard type Time in ESROCOS using unsigned_long (uint_64).
- TimestampConfig: has all the configuration parameters required by the estimator. This is sensor period and the initial latency (if available).
- TimestampStatus: This structure contains all the information regarding the functionality of the Timestamp estimator. This information serves to provide an introspective analysis of the functionality.
- CircularArray: This is a circular or cyclic buffer of fixed size. Such array keeps a window of "size" last samples to perform the estimation of the timestamp, even when no period is provided. The best possible period and timestamp are estimated per each sample, this is done by the TimestampEstimator
- TimestampEstimator: The class computes the estimation of the timestamp. The estimated time is (time_in – latency - jitter). The computation of the latency (when possible) and the jitter is the work for this class.

5.3.5.2. ALIGNMENT MECHANISMS

This module monitors and corrects the alignment of samples arriving to the Provider Interfaces (PI). Since different processing chains have different computation times, it means that, for some critical components, the timestamped data might arrive out-of-order at the components that need it. This issue become more critical since the robot has moving parts (e.g. motors) and the "right" association needs to be made among different channels. Figure 5-28. shows an illustration of this issue with three devices: a servo motor, a LiDAR and a camera.

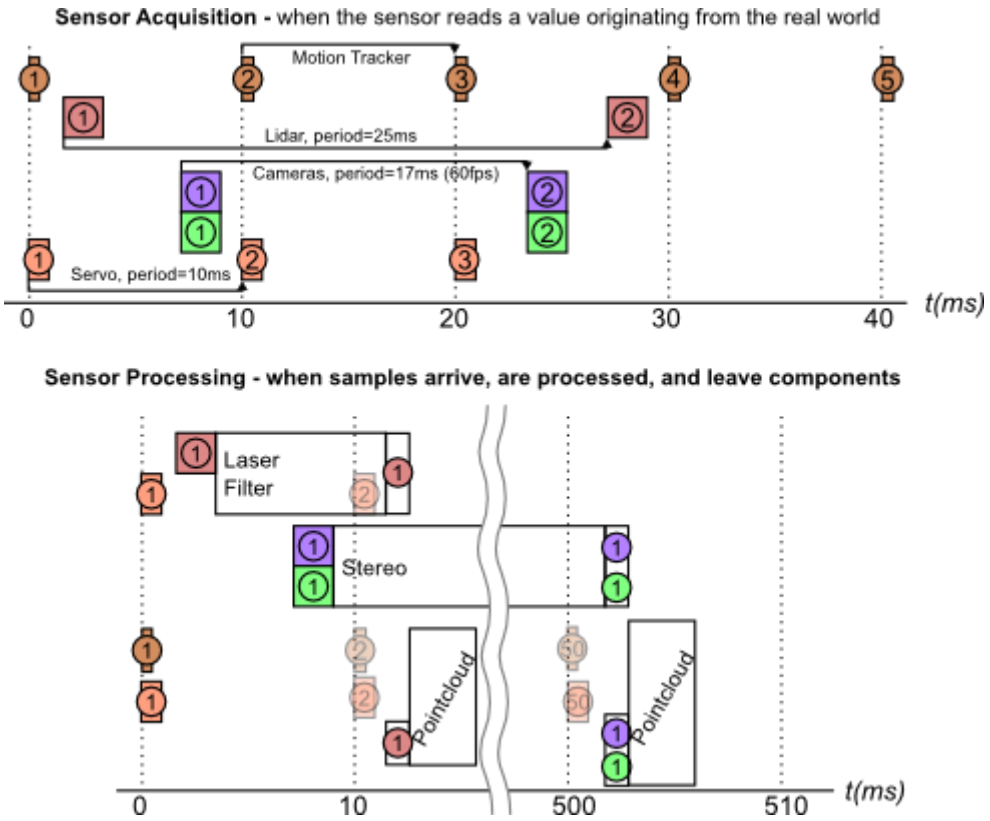


Figure 5-28. The stream alignment issue. Sensor acquisition from the physical world (top). Sensor processing time affecting the practical alignment of samples (bottom)

In this diagram, both times, the pointcloud processing must make sure that data processing is properly associated in time. Moreover, this diagram does not take into account that sensors have different acquisition latencies. It would not be uncommon, in such a processing pipeline, that the first LIDAR sample arrives after the second motion tracker (or servo) sample.

The stream is not an analysis tool as BIP or Cheddar. It is a library to use at runtime to monitor the status and, if possible, align the samples to execute in the right order according to their timestamps. Figure 5-29 illustrates the main working principle:

- The data is pushed as it arrives to the Provider Interface (PI). The leading assumption is that, on each stream, the timestamps are monotonous (they do not go back in time).
- The respective callback is called when it is time to process the relevant samples. This is done only when the stream aligner determined that no sample from other streams could arrive with an earlier timestamp then the one that is being passed to a callback.

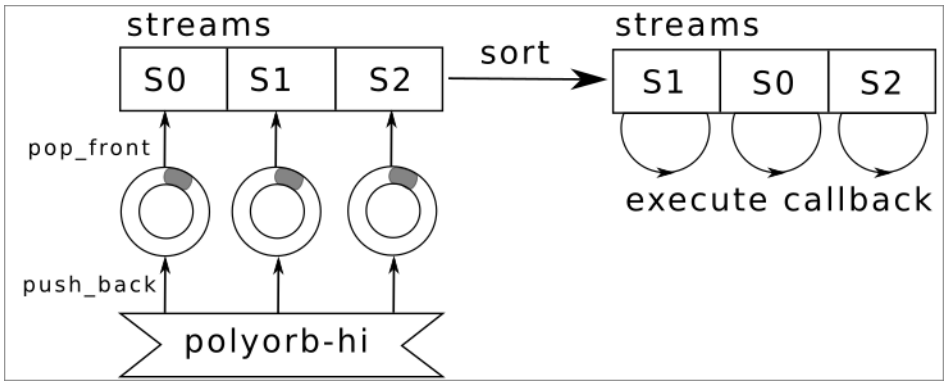


Figure 5-29. Conceptual illustration of the stream aligner mechanism. The samples are queued and processed after sorting according to the timestamp

Because processing based on the stream aligner is based on the fact that samples are passed in-order, the stream aligner must drop samples that arrive "in the past", i.e. that arrive with an earlier timestamp that the last sample processed by the stream aligner.

A UML diagram of the module is given in Figure 5-30. The following individual classes or structures can be identified in the diagram:

- base::Time: This is the standard type Time in ESROCOS using unsigned_long (uint_64).
- CircularBuffer: same concept as in the TimestampEstimator
- Stream: Information to keep per input port or Provider Interface (PI). The class has an instantiation of the circular buffer. This buffer can directly be the TASTE buffer in the PI interface (TBC).
- StreamAligner: This class holds all the streams, one stream per PI, perform the alignment of samples and invokes the call-back function.
- StreamStatus keeps update status of one stream. The following relationship should hold samples_received == samples_processed +samples_dropped_buffer_full + samples_dropped_late_arriving.
- StreamAlignerStatus, information of all modelled streams in the StreamAligner.

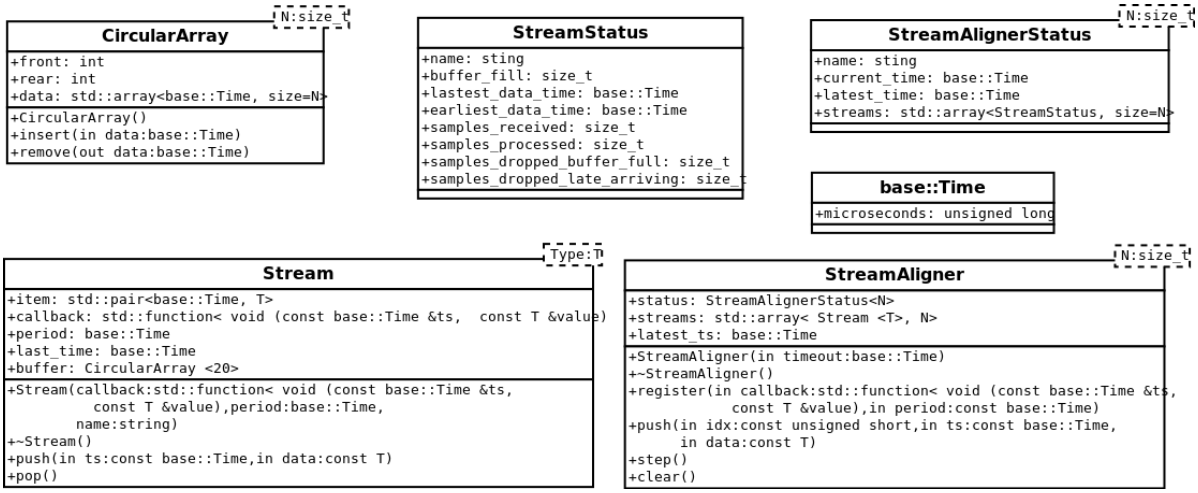


Figure 5-30. Reduced view of the API for the StreamAligner

5.3.6. PUS SERVICES

The Packet Utilization Standard (PUS) , defined in [RD.16], is the standardized TM/TC interface for European space missions. It defines a set of common services that the on-board system provides to the ground segment for the monitoring and control of the space system. Although space robotics applications have different needs with respect to, e.g., satellites, many of the services and concepts defined by the PUS specification are applicable to robotic missions.

For this reason, ESROCOS provides a software library that implements a set of PUS services. The services are implemented at the logical level, i.e., they are representative of the functionality but not of the protocol and formatting of the space data links. The services are implemented in C and optionally provided as TASTE functions, so that they can be integrated in robotics applications modelled with TASTE.

5.3.6.1. STATIC ARCHITECTURE

The static architecture of PUS services library is composed by the data types used in communications defined in ASN.1, the C library which describe the internal behaviour of the services and the capability to missionize the library to use it with different configurations easily.

5.3.6.1.1. ASN.1 TYPES

ASN.1 is the language used to define the different types that are used in telecommands and telemetry packets. TASTE requires that all data sent by its interfaces between different functions be defined in ASN.1, so the use of ASN.1 types is mandatory in this library.

The table below shows the different files, with their short summary, which define all types of data that can be used in TASTE and in a C implementation. When ASN.1 is compiled it generated files in C with all types and constants defined and functions to encode and decode each type.

Table 5-5. PUS Services ASN.1 types files

File name	Brief
ccsds_packet_fields.asn	Definitions for CCSDS packet fields

File name	Brief
ccsds_packet.asn	Definitions for CCSDS packets. Also define the data fields of packets that include another packets like ST12 and ST19.
pus_config.asn	Constant definitions related to the configuration of PUS services
pus_nbit_integers.asn	Definitions for small N-bit integers used in PUS packets
pus_services.asn	Definitions for the standard PUS services and subservices
pus_st01.asn	Definitions for service ST[01] telecommand verification
pus_st03.asn	Definitions for service ST[03] Housekeeping
pus_st05.asn	Definitions for service ST[05] event reporting
pus_st08.asn	Definitions for service ST[08] Function management
pus_st09.asn	Definitions for service ST[09] Time management
pus_st12.asn	Definitions for service ST[12] On-board monitoring
pus_st18.asn	Definitions for service ST[18] On-board control procedure
pus_st19.asn	Definitions for service ST[19] Event-Action
pus_st20.asn	Definitions for service ST[20] On-board parameter management
pus_st23.asn	Definitions for service ST[23] File management
pus_time.asn	Definitions for the time types used in PUS packets

As it is shown in the Table 5-5, there are ASN.1 files for PUS services that need their own data types for different fields in their packets. Also, there are other files that is used for general purposes in the library like ccsds_packet.asn or pus_time.asn.

There are one file, called pus_config.asn, that is used to define constants for the different services, for example, the maximum size of an array or the length of a file.

This is an example of an ASN.1 file, in this case, types specifications for ST[01] Request Verification. The first part of the definition is used to import data types or constant from another module.

Then a data type and different constants are defined. After that, there are some sequence types defined and, in the end, a sequence to define the structure of the data field of TM[1,X] packets.

pus_st01.asn

```
-- ASN.1 definitions for service ST[01] telecommand verification
-- (cf. ECSS-E-ST-70-41C sections 6.1.4.3, 8.1)
--
-- This file defines some error codes for TC verification. Additional error codes
-- can be added


PUS-ST01 DEFINITIONS ::=
BEGIN

    IMPORTS
```



```
    PusPacketVersion, PusPacketType, PusSecondaryHeaderFlag, PusApid, PusSequenceFlags,
PusSequenceCount, PusStepId FROM PUS-CcsdsPacketFields
    PusInt32, PusUInt32, PusUInt64, PusMemAddr FROM PUS-NbitIntegers;

    -- Type to represent failure codes used by service ST[01]
    PusSt01FailureCode ::= PusUInt32

    -- Default failure codes (others may be added)
    pus-ST01-NO-ERROR                   PusSt01FailureCode ::= 0      -- Default value
    pus-ST01-ERROR-APID-UNAVAILABLE     PusSt01FailureCode ::= 1      -- Destination
process not available
    pus-ST01-ERROR-SERVICE-UNAVAILABLE  PusSt01FailureCode ::= 2      -- PUS service
not available
    pus-ST01-ERROR-SUBTYPE-UNAVAILABLE  PusSt01FailureCode ::= 3      -- PUS service
subtype not available
    pus-ST01-ERROR-APID-UNKNOWN         PusSt01FailureCode ::= 4      -- Destination
process is unknown
    pus-ST01-ERROR-SERVICE-UNKNOWN      PusSt01FailureCode ::= 5      -- PUS service
is unknown
    pus-ST01-ERROR-SUBTYPE-UNKNOWN      PusSt01FailureCode ::= 6      -- PUS service
subtype is unknown
    pus-ST01-ERROR-WRONG-FORMAT         PusSt01FailureCode ::= 7      -- Packet format
is inconsistent
    pus-ST01-ERROR-CHECKSUM             PusSt01FailureCode ::= 8      -- Checksums are
currently not used, code reserved for future use

    -- Type to report failure data in ST[01] TM messages
    -- The type matches the error information stored by the PUS library (pus_error.h).
Fields will be
    -- used optionally, depending on the code.
    PusSt01FailureInfo ::= SEQUENCE
    {
        subcode PusInt32,       -- Internal error code, matching a C enum
        data    PusInt32,       -- Error data
        address PusMemAddr      -- A memory address
    }

    -- Request ID, used by ST[01] packets
    PusSt01RequestId ::= SEQUENCE
    {
        packetVersion       PusPacketVersion,
        packetType          PusPacketType,
        secondaryHeaderFlag PusSecondaryHeaderFlag,
        apid                PusApid,
        sequenceFlags       PusSequenceFlags,
        sequenceCount       PusSequenceCount
    }

    -- Failure notice, used by ST[01] packets
    PusSt01Failure ::= SEQUENCE
    {
        code    PusSt01FailureCode,
        info    PusSt01FailureInfo
    }

    -- Data of ST[01] TM packets
    -- Each packet type uses a subset of the fields, but it has been preferred to keep
    -- the same structure for all to reduce the amount of code
    PusTM-1-X-Data ::= SEQUENCE
    {
        request PusSt01RequestId,
        step    PusStepId,
        failure PusSt01Failure
    }
END
```

5.3.6.1.2. C LIBRARY

The Table 5-6 enumerates the components of the PUS Services C library. There are components to manage the creation of the different packets of each service, other

manage the internal functionality of each service and others general functionality for all the services implemented.

Table 5-6. PUS Services C library files

File name	Brief
pus_apid.h/.c	Data type to handle the TM count of an Application Process
pus_error.h/.c	Functions to record internal error conditions
pus_event_action.h/.c	Management of the event-action definitions table
pus_events.h/.c	Management of the events table used by several services
pus_file_management.h/.c	File management
pus_housekeeping.h/.c	Management of the housekeeping and diagnostics parameters table
pus_obcp_engine.h/.c	Management of the on-board control procedures engine
pus_packet.h/.c	Functions for managing PUS TM/TC packets and their header information
pus_packet_queues.h/.c	Functions for managing PUS TM/TC packets queues
pus_packet_reduced.h/.c	Packet reduced functions
pus_parameter_management.h/.c	Management of on-board parameters
pus_parameter_monitoring.h/.c	Management of the parameter monitoring table
pus_st01_packets.h/.c	PUS service ST[01] Request Verification
pus_st03_packets.h/.c	PUS service ST[03] Housekeeping
pus_st05_packets.h/.c	PUS service ST[05] Event Reporting
pus_st08_packets.h/.c	PUS service ST[08] Function management
pus_st09_packets.h/.c	PUS service ST[09] Time management
pus_st11_packets.h/.c	PUS service ST[11] Time-based scheduling
pus_st12_packets.h/.c	PUS service ST[12] On-board Monitoring
pus_st17_packets.h/.c	PUS service ST[17] Test Connection
pus_st18_packets.h/.c	PUS service ST[18] On-board control procedure
pus_st19_packets.h/.c	PUS service ST[19] Event-action
pus_st20_packets.h/.c	PUS service ST[20] On-board parameters management
pus_st23_packets.h/.c	PUS service ST[23] File management
pus_stored_param.h/.c	Management of the Stored parameters in packets
pus_threads.h/.c	Multi-threading functions used by the PUS library
pus_time.h/.c	Functions to read the on-board time and convert to/from POSIX time
pus_timebased_scheduling.h/.c	Time-based Scheduling functionality
pus_types.h	Types used in the PUS packet data structures

The functionality of the PUS services are specified in the ECSS standard [RD.16]. In the frame of ESROCOS, a subset of the capabilities defined in the standard has been implemented at logical level (i.e., the logic of the services is implemented, but the packet and transport details are not representative and rely on TASTE instead).

The following paragraphs detail which services and capabilities are implemented, and refer to the standard for the details of each service. At source code level, the function interfaces are documented in Doxygen and available separately.

ST[01] Request Verification

The Request Verification service, specified in sections 6.1 and 8.1 of [RD.16], defines three subservice types: routing reporting, acceptance reporting and execution reporting. This library implements the acceptance and execution reporting subservices.

These subservice includes the capability to report the success or failure of the acceptance, start of execution, progress of execution and completion of execution of a request received by the APID where is service is allocated.

These functionalities are implemented by the creation of eight telemetry packets, two per each functionality, respectively for the success report and failure reports. Each packet type uses a subset of the fields, but it has been preferred to keep the same structure for all to reduce the amount of code.

Failure reports have a field in which user can specify the type of error and other parameters related to this error.

ST[03] Housekeeping

The Housekeeping service has three subservices defined in sections 6.3 and 8.3 of [RD.16]: housekeeping reporting, diagnostic reporting and parameter functional reporting configuration. Only housekeeping reporting is implemented and only with the functionality of TM[3,25] Housekeeping report.

With this service the library provides the capability to report values of each on-board parameter that is accessible to the application process that hosts that service. User can define a report in JSON configuration file with the list of parameter that the report is going to contain.

The next code shows an example of the definition of parameters of each mission. *Label* and *type* fields are for this service, meanwhile *low_limit* and *high_limit* fields are from ST[12] On-board Monitoring.

st03_st12_config.json

```
"parameters": [
      {
            "label": "HK_PARAM_INT01",
            "type": "INT32",
            "low_limit": 4,
            "high_limit": 10
      },
      {
            "label": "HK_PARAM_REAL01",
            "type": "REAL64",
            "low_limit": -23,
            "high_limit": 23.23
      },
```

Each report contains the number of parameter reported, an array with the identifiers of each parameter and, of course, the value of each parameter.

All housekeeping parameters are reported in an array of 64 bit elements, regardless of their actual size. The length of the housekeeping report is defined by a constant, not derived from the mission database.

There is a table that contains the values of each on-board parameter and a table that contains information of them. This information include two fields; label, used to identify each parameter, and the data type of the parameter.

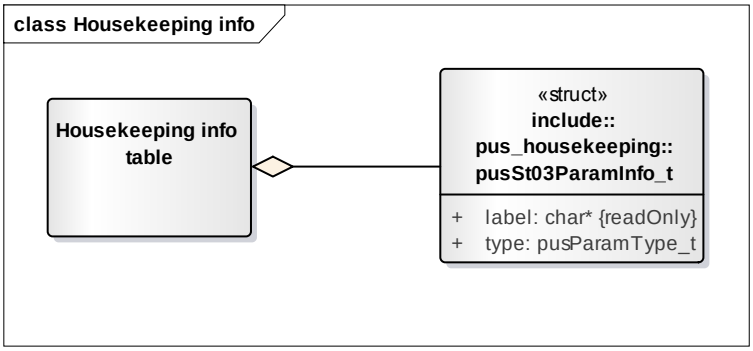


Figure 5-31. PUS Housekeeping info table

Those tables are initialized with the service and load the information from the JSON configuration file of the service where user has to set the size of those tables and the field need for info table.

ST[05] Event Reporting

The Event Reporting service, specified in sections 6.5 and 8.5 of [RD.16], is an important service in the library, it provides a table of the different events that the spacecraft can generate. ST[18] and ST[19] also access to this table that is loaded when ST[05] is initialized.

This service has a JSON configuration file where user defines the different events that can be loaded in the events table, the maximum size of a circular buffer where events that the spacecraft generate are registered, and the destination of the event reports generated by this service.

The event info table is composed by an array of event info structs, which have the event with its types of data and values. All events use the same structure for their auxiliary data field.

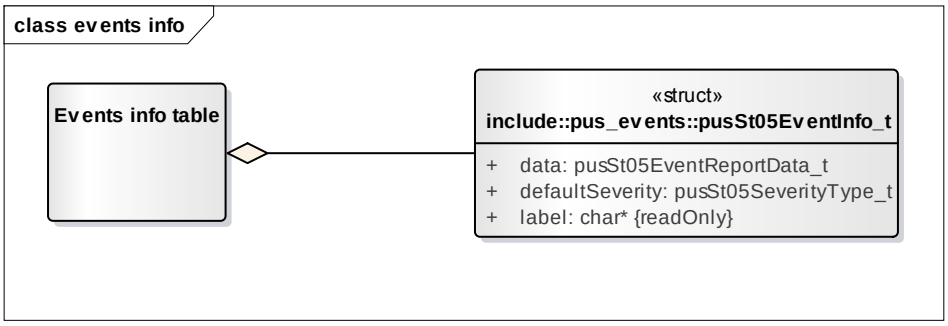


Figure 5-32. PUS Events info table

If an on-board event occurs, it is pushed into a circular buffer where the different services can access it. If the buffer is full of events, the first of them that was pushed is deleted to let space to a new event. The access to this circular buffer can be protected by mutex of the pthread C library or by TASTE own mechanisms.

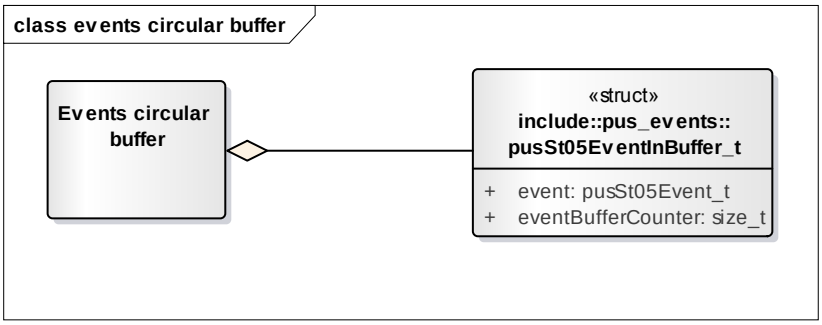


Figure 5-33. PUS Events circular buffer

Each service that need to access to this buffer has a counter to know which event was the last that it read. The event service has a function to get the next event depending on the value of this counter so that a service request the next service and has the capability to know if it has missed one event or not.

ST[08] Function management

The Function Management service has only one subservice as is defined in sections 6.1 and 8.1 of [RD.16], perform a function. It is a telecommand which contains a field that is an identifier of the function that the user requests to execute on-board.

This telecommand does not accept arguments for functions despite the standard specify that each function can have an array of arguments. The service ST[12] can be used to achieve the equivalent functionality.

This service has to be initialized before using it because the library has to load, code generated to the mission from the JSON configuration file of the service, the different functions that can be performed. Different functions are provided to read the table, locate and execute the functions.

Next code is an example of JSON configuration scheme file where a functions array is defined with two strings fields called label and name.

pus_st08_scheme.json

```
{
    "$schema": "http://json-schema.org/schema#",
    "id": "https://github.com/esrocos/TBD",
    "title": "On-board functions",
    "description": "Functions defined for ST08",
    "type": "object",
    "properties": {
       "functions": {
            "type": "array",
            "items": {
                "title": "Function",
                "type": "object",
                "properties": {
                    "label": {
                        "description": "Textual label for the function",
                        "type": "string"
                    },
                    "name": {
                        "description": "Name of the function",
                        "type": "string"
                    }
                },
                "required": ["label", "name"]
            }
        }
    },
    "required": ["functions"]
}
```

ST[09] Time management

The Time Management service, specified in sections 6.9 and 8.9 of [RD.16], includes two subservices, time reporting and time reporting control, and both are implemented. Time reporting has two time code format, CUC and CDS, being CUC chosen for this library implementation.

The time reporting control subservice manages the rate of generation of the time reports. There is a telecommand associated to his subservice to set the exponential rate from 1 to 8, so that the rates for time report generation are 1, 2, 4, 16, …, 256. If a rate of N is set, the service sends one time report every N telemetry packets sent from the application process. Time packets use the APID 0.

There is no JSON configuration file nor the need of initialize this service.

ST[11] Time-based scheduling

The Time-Based Scheduling service, specified in sections 6.11 and 8.11 of [RD.16], only has one subservice, with the same name, that includes the capability to maintain an on-board time-based schedule of requests and to ensure their timely release.

Not all the functionalities of this service are implemented, only a subset of them: enable, disable and reset service, and add activities to the schedule. Apart from the rest of telemetry and telecommands functionalities that are not developed, there are other functionalities link sub-schedule and group of task that are also not implemented.

The addition of multiple tasks per request are available as specified by the standard. The data kind available for this tasks are a subset of all data kinds to prevent that a task that is a telecommand include another one and it makes an infinite cycle.

There is a JSON configuration file where the user can define the maximum number of activities that can be in the time-based schedule table. This table has to be initialized with the service and it has to reject all requests if the service is not initialized.

This service is developed in two parts; one is the creation of the telecommand packets, and the other is the management of the time-based scheduled table.

Among the implemented packets, only the telecommand TC[11,4] Insert activity has data field which include the list of tasks and the time when they should be executed.

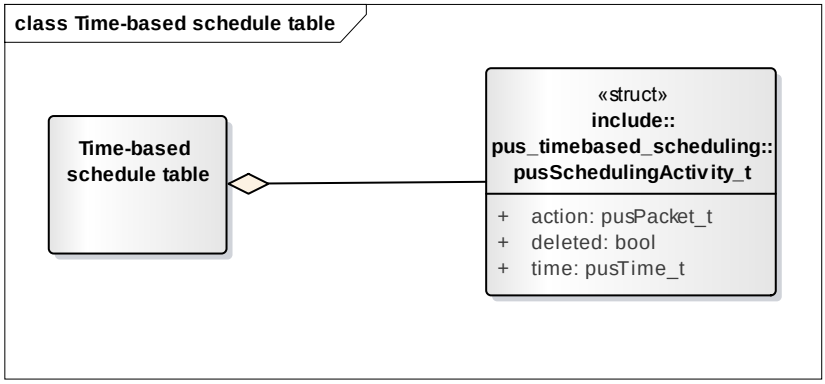


Figure 5-34. PUS time-based schedule table

As the above figure shows, the table is composed by an array of structures that contains the task in TC packet form, the time to execute it and a Boolean variable to know the status of this activity.

ST[12] On-board Monitoring

The On-board Monitoring service, specified in sections 6.12 and 8.12 of [RD.16], defines two subservices, parameter monitoring and functional monitoring, but only parameter monitoring is implemented. A subset of functions are implemented TC[12,1] enable parameter monitoring definitions, TC[12,2] disable parameter monitoring definitions, TC[12,15] enable parameter monitoring and TC[12,16] disable parameter monitoring.

This service has to be initialized to load a table that store the limits between the parameter values should be. If a value is out of this range, the service generates an event and sends it to the events table, created in ST[05] Event Reporting.

There is a component that manage the creation of the TC[12,X] packets and other that manage the on-board parameter monitoring table, enabling or disabling definitions or the general service.

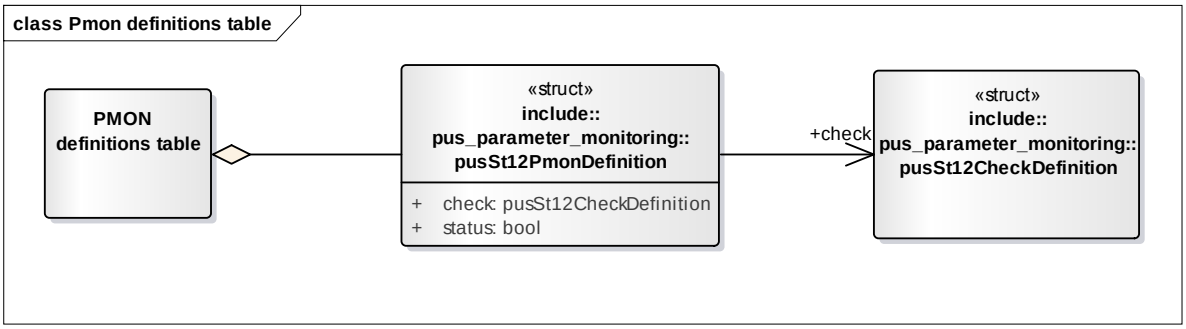


Figure 5-35. PUS PMON definitions table

The figure above represents the PMON (parameter monitoring) definition table, it is an array of PMON definitions that include another structure, which define the limits for each parameter, and the status of each definition.

ST[17] Test Connection

Test Connection service, specified in sections 6.17 and 8.17 of [RD.16], is used to check if there is a connection between the ground station and the spacecraft. The standard specifies an "are-you-alive" and an "on-board connection test" subservice.

In this case, only are-you-alive connection test is implemented with its telecommand and telemetry packets, TC[17,1] and TM[17,2]. When the ground station sends a request of the test, the spacecraft has to send the report for this request.

This service has no JSON configuration file, neither the need of a initialization because there are no parameters to load.

ST[18] On-board Control Procedure

The On-board Control Procedure service, specified in sections 6.18 and 8.18 of [RD.16], is one of the most important and complex services in this library. It has two subservices defined, OBCP management and OBCP engine management, and both of them are implemented. OBCP behaviour is defined in [RD.24].

The OBCP engine management is fully implemented, however, OBCP management has only a subset of function implemented; load OBCP direct or by reference and the capability to unload, activate, stop, suspend, resume and abort OBCP procedures.

The programming language selected for OBCP coding is MicroPython [RD.35], which is an efficient implementation of the Python 3 programming language with a small subset of its standard libraries and is optimised to rum on microcontrollers and constrained environments.

ST[19] Event-Action

The Event-Action service, specified in sections 6.19 and 8.19 of [RD.16], uses the events table initialized by ST[05] Event Reporting that reads and do the action defined for each event if is needed. The development of this service is divided in two parts; the packet creation with the getters and setters of each fields, and the management of the events-actions definitions table.

Only a subset of subservices are implemented. There are telecommands to add, delete, enable and disable an event-action definition. All of them have the limitation that can only control one definition per telecommand.

This service creates also a table, in this case is an event-action definitions table, that is used to perform an action when an event has appeared in the events table and it has an event-action definition defined and activated. This library only allows for one event-action definition per event.

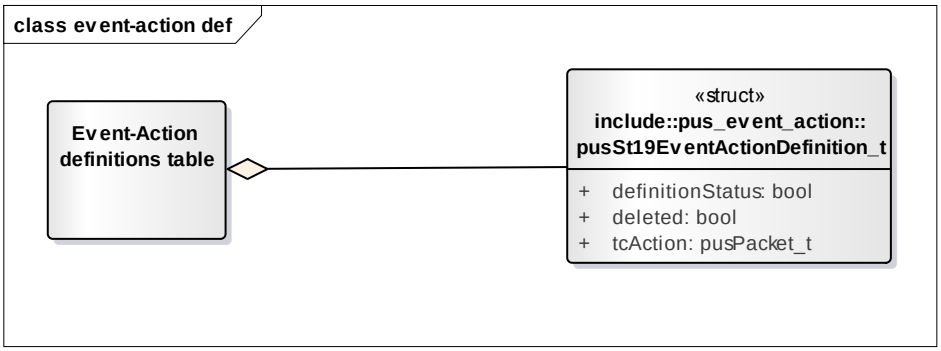


Figure 5-36. PUS Event-Action definitions table

The figure above shows the structure of the event-action definitions table, it is an array of structures that contain the status of the definition and the action associated to each definition.

ST[20] On-board Parameters Management

The On-board Parameters Management service, specified in sections 6.20 and 8.20 of [RD.16], provides the capability for managing on-board parameters, different from housekeeping ones, including reading and setting values. Not all the functionalities are implemented, only request and report a parameter value and set a new one is available. Reports of parameters definitions and memory address changes are nor implemented.

Like in all the services there are a pair of components that create the packets needed for this service. In this case, there are only three packets TC[20,1] request a parameter value, TM[20,2] report a parameter value and TC[20,3] set a parameter value.

This service needs to be initialized to load two tables with the size specified in the JSON configuration file. Those tables are like the ST[03] Housekeeping ones, one is for the parameter information (label and data type) and the other one is for the values stored in 64 bits, with is an array of unsigned integer in 64 bits. The identification of each parameter is the index in the tables.

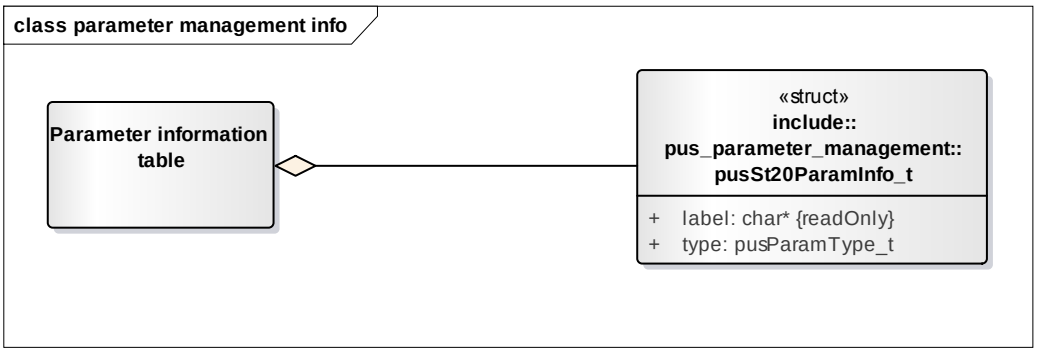


Figure 5-37. PUS Parameter management information table

There are functions that convert this stored values in 64 bits to its data types looking in the information table of the service and in the other way around. When the service is initialized, some C functions are created to get and set the value of each parameter with the appropriate data type.

ST[23] File management

File management service, specified in sections 6.23 and 8.23 of [RD.16], is another important service in the library. The standard defines two subservices the file handling subservice and the file copy subservice, both of them are implemented but with a subset of functions, not all of them.

In file handling subservice create file, delete file and report attributes functions are defined and in file copy service only the copy function is also defined. Neither the possibility of blocking files nor the addition of additional attributes are implemented in this library.

5.3.6.1.3. MISSIONIZATION

Most of services have a JSON configuration file, this file is where the user of the library can set different parameters to generate a mission database specific for an application. User can set events information, functions, size of vectors, etc., according to the configuration of each service.

Each JSON configuration file has a JSON schema file associated to the same service. This scheme is used to check that the configuration is corrects and the user set the parameters that is needed and in the correct format.

st03_st12_schema.json

```
{
    "$schema": "http://json-schema.org/schema#",
    "id": "https://github.com/esrocos/TBD",
    "title": "HK parameters",
    "description": "Housekeeping parameters configuration for PUS service ST[03] and
ST[12]",
    "type": "object",
    "properties": {
        "parameters": {
            "type": "array",
            "items": {
                "title": "HK parameter",
                "type": "object",
                "properties": {
                    "label": {
                        "description": "Textual label for the parameter",
                        "type": "string"
                    },
                    "type": {
                        "description": "Data type of the parameter",
                        "type": "string",
                        "enum": [
                            "INT32",
                            "UINT32",
                            "REAL64",
                            "BYTE",
                            "BOOL"
                        ]
                    },
                    "low_limit": {
                        "description": "Low limit for PMON",
                        "type": "number"
                    },
                    "high_limit": {
                        "description": "High limit for PMON",
                        "type": "number"
                    }
                },
                "required": [
                    "label",
                    "type",
                    "low_limit",
                    "high_limit"
```


st03_st12_schema.json

```
                ]
            }
        },
        "defaultHkReport": {
            "type": "array",
            "items": {
                "description": "Textual label for the parameter, matching the parameters
array",
                "type": "string"
            }
        }
    },
    "required": [
        "parameters",
        "defaultHkReport"
    ]
}
```

st03_st12_config.json

```
{
       "parameters": [
              {
                     "label": "HK_PARAM_INT01",
                     "type": "INT32",
                     "low_limit": 4,
                     "high_limit": 10
              },
              {
                     "label": "HK_PARAM_REAL01",
                     "type": "REAL64",
                     "low_limit": -23,
                     "high_limit": 23.23
              },
              {
                     "label": "HK_PARAM_INT02",
                     "type": "INT32",
                     "low_limit": 4,
                     "high_limit": 10
              },
              {
                     "label": "HK_PARAM_UINT01",
                     "type": "UINT32",
                     "low_limit": 5,
                     "high_limit": 10
              },
              {
                     "label": "HK_PARAM_BYTE01",
                     "type": "BYTE",
                     "low_limit": 2,
                     "high_limit": 10
              },
              {
                     "label": "HK_PARAM_BOOL01",
                     "type": "BOOL",
                     "low_limit": 0,
```


st03_st12_config.json

```
                        "high_limit": 0
                }
        ],
        "defaultHkReport": [
                "HK_PARAM_REAL01",
                "HK_PARAM_INT01",
                "HK_PARAM_UINT01",
                "HK_PARAM_BYTE01",
                "HK_PARAM_BOOL01"
        ]
}
```

This check is made through python with the *jsonschema* module and the *validate* method as is shown in the code attached below. This is an example of the service ST03 where schemaFile and missionFile are loaded and then try to validate them.

st03_config.py (25-33)

```
# Load and validate configuration
print(' - generate configuration for service ST[03]')
schema = loadJson(schemaFile)
configData = loadJson(missionFile)
try:
    jsonschema.validate(configData, schema)
except Exception as err:
    perror('Error in ST[03] service configuration {}:\n{}'.format(missionFile, err))
    raise
```

When the schema is validated, the next step is the transformation of the data in JSON to the C language to be compatible with the library. It is realized by a module in python called *mako*.

Mako is used to load the JSON information in a Python dictionary and allow the access to this variables in mako files. Mako is a python library that generates text files, for example C files, from text templates with macros and python methods .After that, these mako files are rendered and create a string depending on the commands lines written in mako files.

st03_config.py (35-42)

```
# Render templates
try:
        template = Template(filename=headerTemplate)
        outHeader = template.render(config=configData,tempvars=dict())
except:
        perror('Error generating {}:'.format(headerTemplate))
        perror(text_error_template().render())
        raise
```

Then, python script copies the string generated, after rendering the mako files, to the source (.c) and header (.h) files.

st03_config.py (52-60)

```
# Write file
try:
        if not os.path.exists(os.path.dirname(outHeaderFile)):
                os.makedirs(os.path.dirname(outHeaderFile))
        with open(outHeaderFile, 'w') as fd:
                fd.write(outHeader)
except Exception as err:
        perror('Error writing ST[03] generated code to {}:\n{}'.format(outHeaderFile, err))
        raise
```

A mako example of use is shown below, it is a mako template to generate a header file in C.

pus_st03_config.h.mako

```
#ifndef PUS_ST03_CONFIG_H
#define PUS_ST03_CONFIG_H

#ifdef __cplusplus
extern "C" {
#endif


#include "pus_error.h"
#include "pus_types.h"
#include "pus_housekeeping.h"


// Parameter identifiers
<% count = 0 %>
% for param in config['parameters']:
#define ${param['label']} ((pusSt03ParamId_t) ${count}) \
<% count = count + 1 %>
% endfor
#define PUS_ST03_PARAM_LIMIT ((pusSt03ParamId_t) ${count})
```

And the result after rendering this mako file is below, there are some defines that used a python counter defined in the mako file and labels from de dictionary generated from JSON configuration files:

pus_st03_config.h

```
// PUS service ST[03] configuration
//
// File automatically generated from the pus_st30_config.h.mako template
//
//                      -- DO NOT MODIFY --

#ifndef PUS_ST03_CONFIG_H
#define PUS_ST03_CONFIG_H
```

pus_st03_config.h

```
#ifdef __cplusplus
extern "C" {
#endif


#include "pus_error.h"
#include "pus_types.h"
#include "pus_housekeeping.h"


// Parameter identifiers

#define HK_PARAM_INT01 ((pusSt03ParamId_t) 0)
#define HK_PARAM_REAL01 ((pusSt03ParamId_t) 1)
#define HK_PARAM_INT02 ((pusSt03ParamId_t) 2)
#define HK_PARAM_UINT01 ((pusSt03ParamId_t) 3)
#define HK_PARAM_BYTE01 ((pusSt03ParamId_t) 4)
#define HK_PARAM_BOOL01 ((pusSt03ParamId_t) 5)
#define PUS_ST03_PARAM_LIMIT ((pusSt03ParamId_t) 6)
```

The generated files are part of the library, their functions, constants, defines or variables are used by other files to create arrays with a specific size, initialize a service with specified parameters, etc.

5.3.6.1.4. OBCP LANGUAGE

MicroPython, [RD.35], is the language used to define the on-board procedures which is an efficient implementation of the Python 3 programming language, it is optimised to run on microcontrollers and in constrained environment and has been ported to LEON [RD.36].

Each OBCP process will be a python class with it owns methods. Each OBCP is divided in several steps, that steps has to be executed entirely before check if the process has to be stopped or cancelled.

OBCPs can invoke, through python functions, methods from the PUS services library, for example; event management, get/set housekeeping parameters, get/set on-board parameters, perform functions, launch another OBCP, etc.

5.3.6.2. DYNAMIC ARCHITECTURE

A reference TASTE implementation has been defined to test the PUS services library and provide a model to use the services in a robotics application. This reference implementation, which is one example of multiple possible implementations, is explained in this section.

As Figure 5-38 shows, the problem has been divided in two parts, ground station and on-board application.

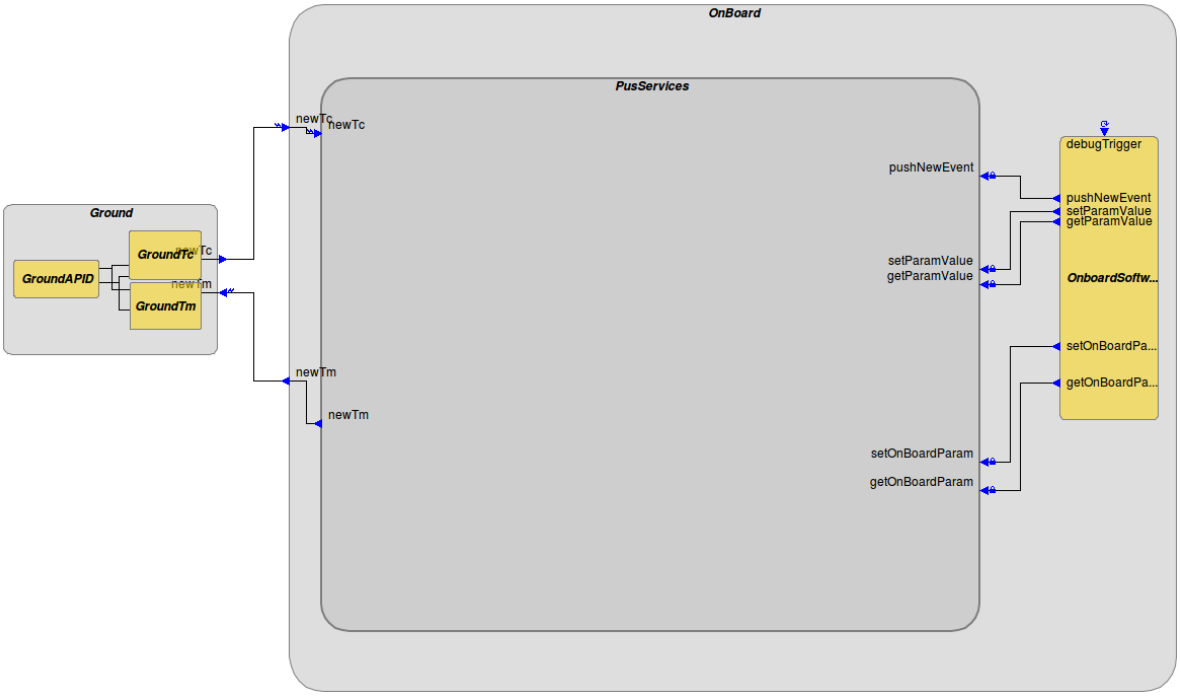


Figure 5-38. Reference PUS implementation in TASTE – general view

The Ground component is composed by the manager of its APID and a manager for telecommand and telemetry packets. It has two interfaces, one provider to send a new telecommand packet and other required to receive a new telemetry packet.

The On-board function has a function to manage the PUS services and another that represents the on-board software that set and get parameters and events from the PUS services library.

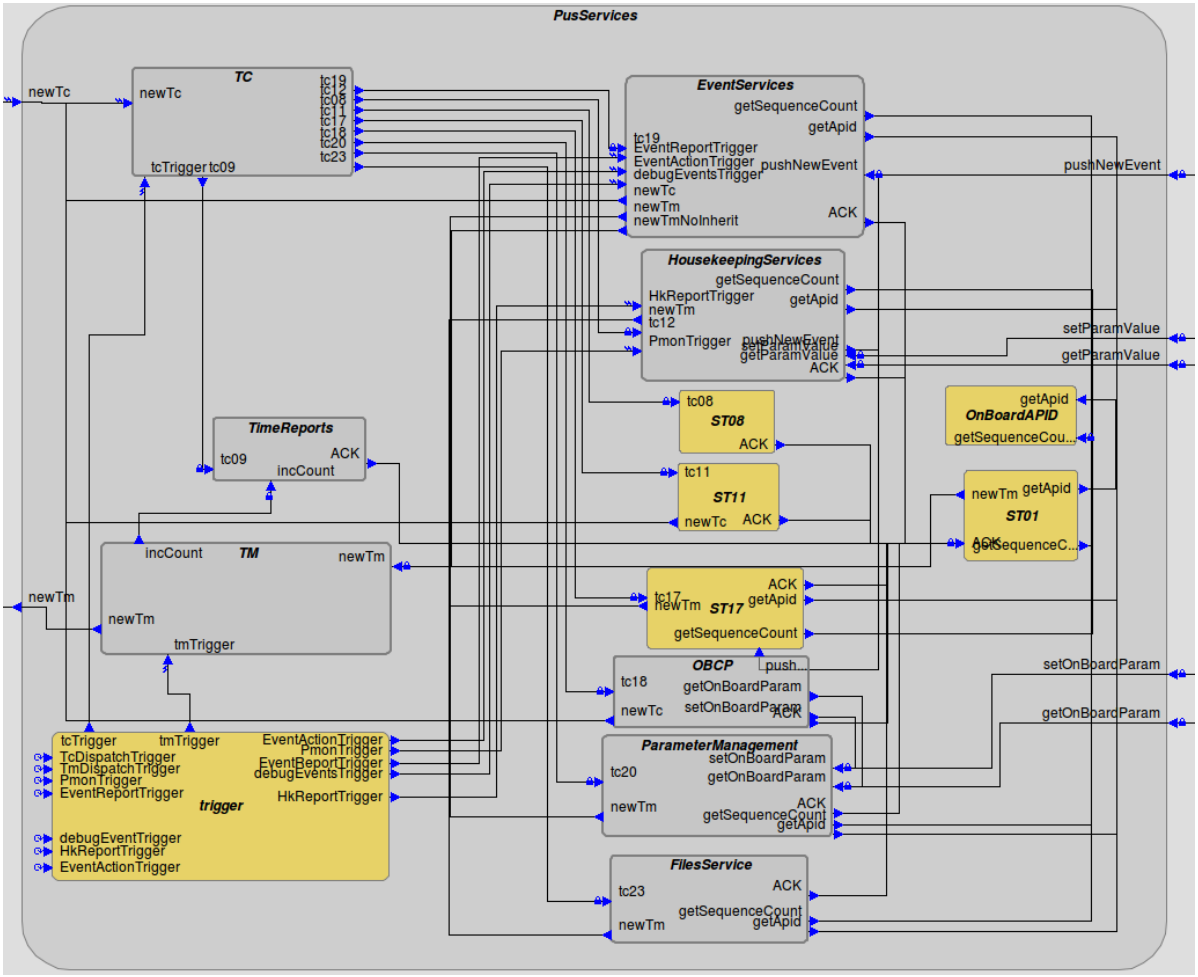


Figure 5-39. PUS Services TASTE component

Each function of these figures is explained in detail below.

The Ground application, as Figure 5-40 represents, has three functions:

- GroundTc: function that creates and sends telecommand packets to a required interface.
- GroundTm: function that receives and reads telemetry packets from a sporadic provided interface.
- GroundAPID: function that manages the application process of the ground application. It has two provided interfaces where the previous functions access to get the application process identifier and the sequence counter for new packets.

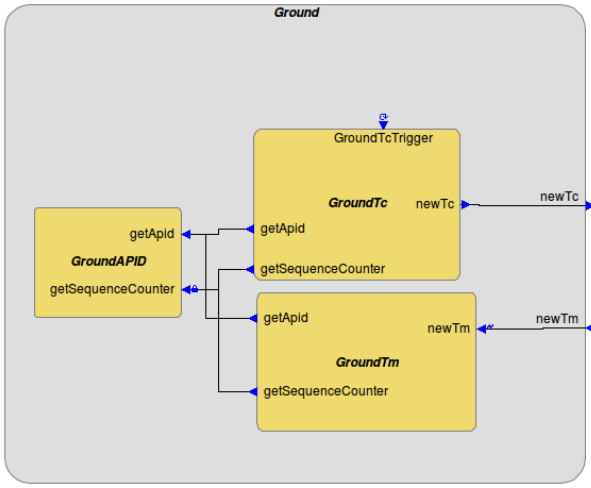


Figure 5-40. Ground component

In this implementation, each task has its own trigger allowing user to set each period of execution easily. Trigger module, as Figure 5-41 shows, has some provided interfaces, all of them are periodic and each one has a required interface associated.

The required interfaces are connected to sporadic provided interfaces in services functions, each service that need to check parameters, events or another periodic task has a trigger to do these actions.

The reference implementation centralised all the triggers as cyclic interfaces for convenience. Applications should trigger the execution of each service according to their scheduling approach.

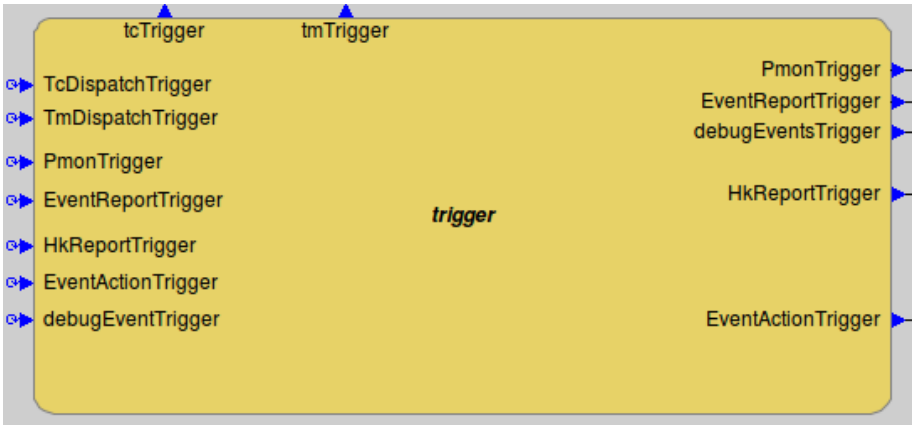


Figure 5-41. Service triggers

TC on-board module, see Figure 5-42, is composed by the TC queue and the TC dispatch. The queue receives and stores the telecommands from de ground segment or a service that generate telecommands.

TC dispatch sends the telecommand requests to the queue and if there is an available telecommand, it is sent to the dispatch to analyse the type of packets and send it to the destination service.

This module has nine required interfaces one per service that have telecommands, and two provided interfaces, one for a new telecommands into the module and the other to manage when to dispatch a request for a new TC. Both provided interfaces are sporadic.

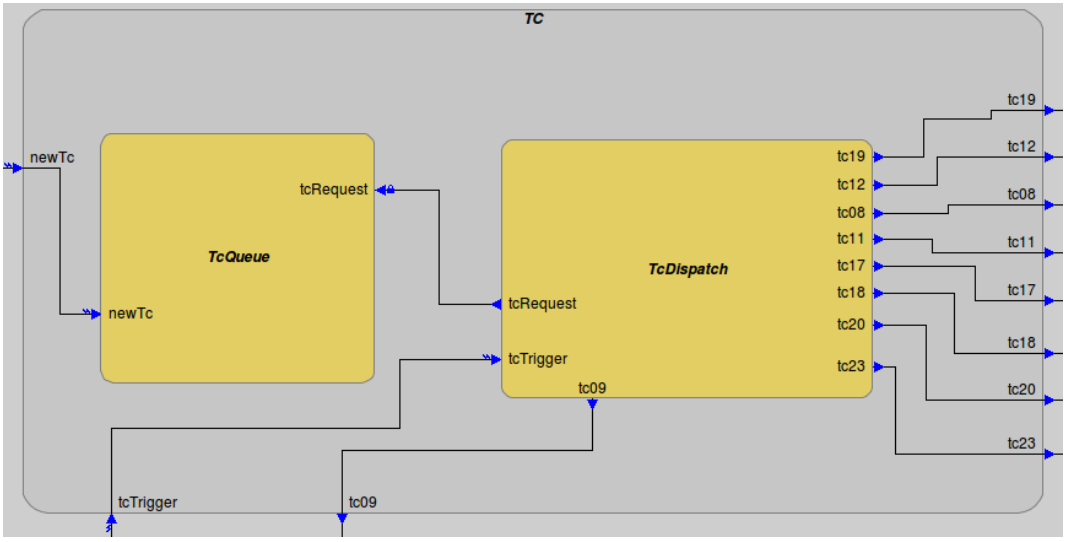


Figure 5-42. TC on-board module

The TM on-board module, Figure 5-43, is pretty similar to TC module. There is a queue and a dispatch function. TM packets are pushed into the TM queue and the TM dispatch requests to get new packets to send to the ground station.

This module has two required interfaces, one is to send TM packets to ground and the other is to manage the number of telemetry packets that has arrived to the queue, this information is used by the ST[09] Time management service.

It has also two provided interfaces, one is to receive news telemetry packets from the services and the other is a trigger to manage the time when the dispatch does the request to send packets to ground station.

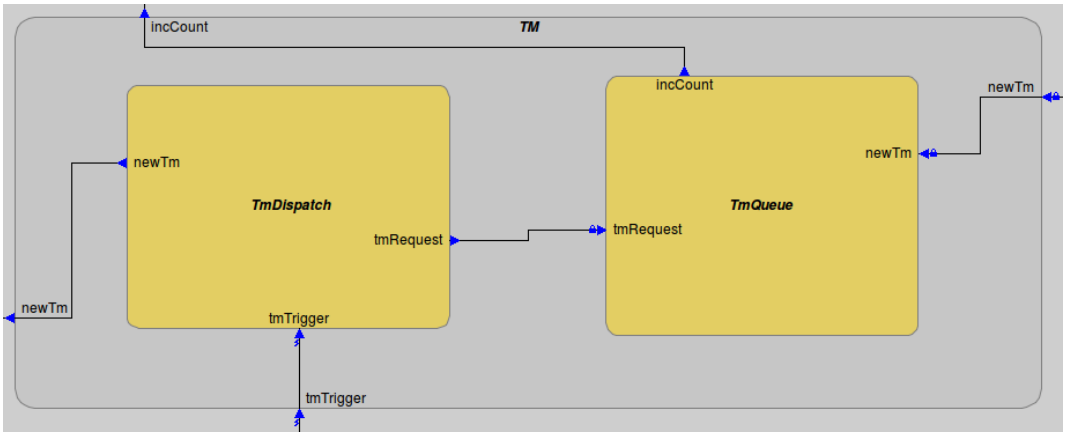


Figure 5-43. TM on-board module

Time reports module, see Figure 5-44, manage the ST[09] Time management with its own APID. The module has two provided interfaces, both protected, one is to receive TC[9,X] packets and the other, *incCount*, is to increment the counter and check if a time report is needed.

There is also a required interface, called *ACK*, that is used to send the different types of acknowledgement to ST[01] Request Verification.

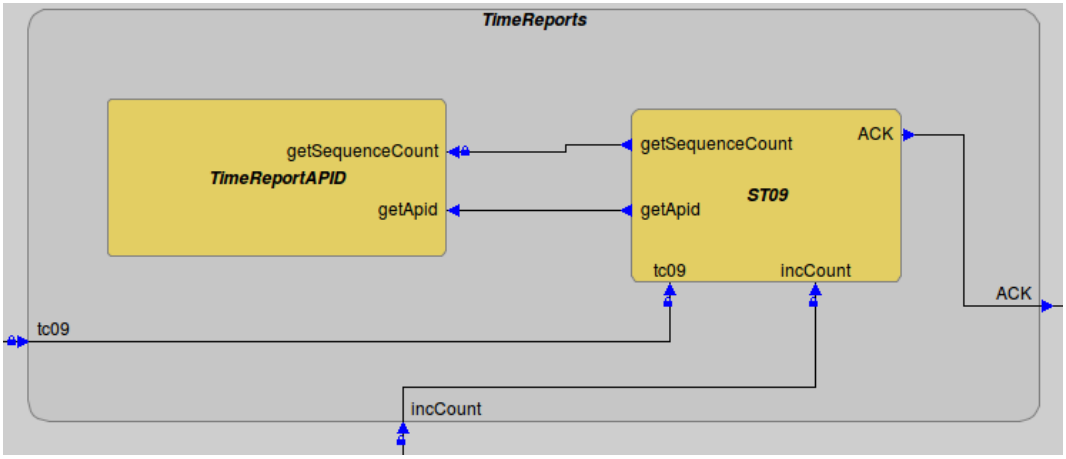


Figure 5-44. Time reports module

The Event Services module is composed by ST[05] Event Reporting, ST[19] Event-Action, and a function to manage the events table with interfaces that let services to get events and push new events.

The module has five provided interfaces:

- Tc19: protected, new telecommands for ST[19] Event-Action.
- EventReportTrigger: sporadic, trigger to check if there is an event that need a report.
- EventActionTrigger: sporadic, trigger to check if there is an event with a definition enabled.
- debugEventsTrigger: sporadic, trigger used for debug implementation, to insert events in table from time to time.
- pushNewEvent: protected, new event for events table.

And five required interfaces:

- newTc: telecommands packet to TC queue.
- newTm: telecommands packet to TM queue.
- newTmNoInherit: same as *newTm* but duclicated because the bug of multiple interfaces in functions.
- getSequenceCount: get sequence counter from on-board APID.
- getApid: get APID from on-board APID
- ACK: send the different types of acknowledgement to ST[01] Request Verification.

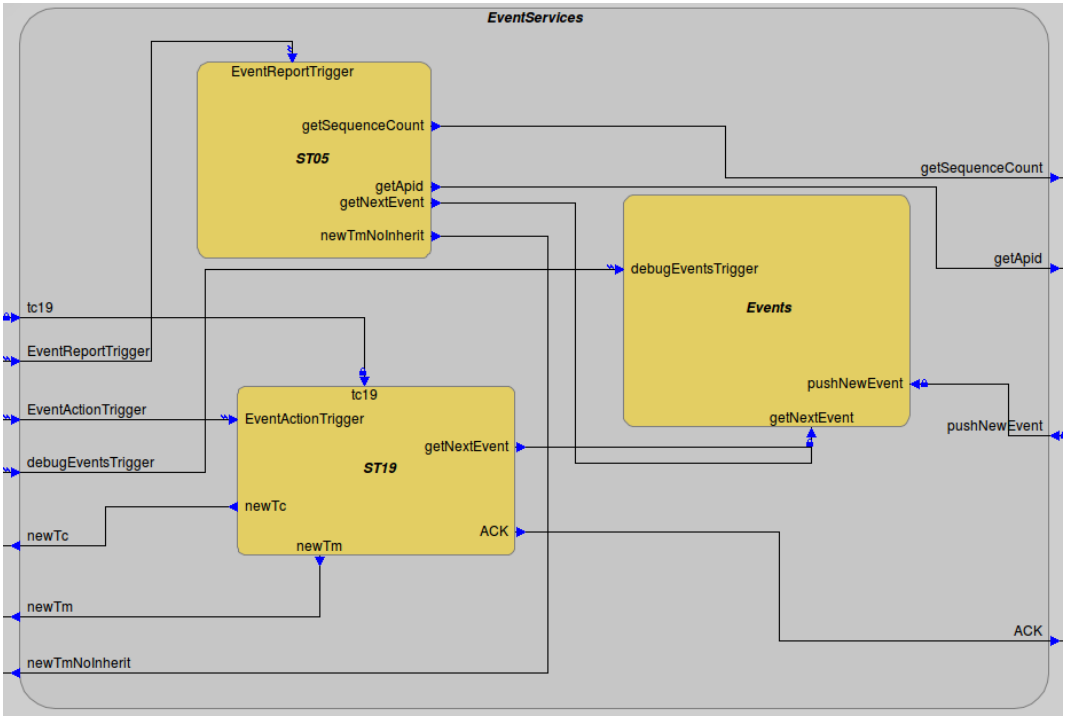


Figure 5-45. Event Services module

The Housekeeping Services module is similar to the Event Services module. It includes ST[03] Housekeeping, ST[12] On-board Monitoring and a function that manages housekeeping parameters.

Provided interfaces:

- tc12: protected, new telecommands for ST[12].
- HkReportTrigger: sporadic, trigger to perform the creation of a housekeeping report.
- PmonTrigger: sporadic, trigger to check in housekeeping table if any parameter has a value out of range to create a report.
- setParamValue: protected, set parameter value in 64 bits.
- getParamValue: protected, get parameter value in 64 bits.

Required interfaces:

- newTc: telecommands packet to Tc queue.
- newTm: telemetry packet to TM queue.
- getSequenceCount: get sequence counter from on-board APID.
- getApid: get APID from on-board APID
- pushNewEvents: protected, new event for events table from an event-action definition.
- ACK: send the different types of acknowledgement to ST[01] Request Verification.

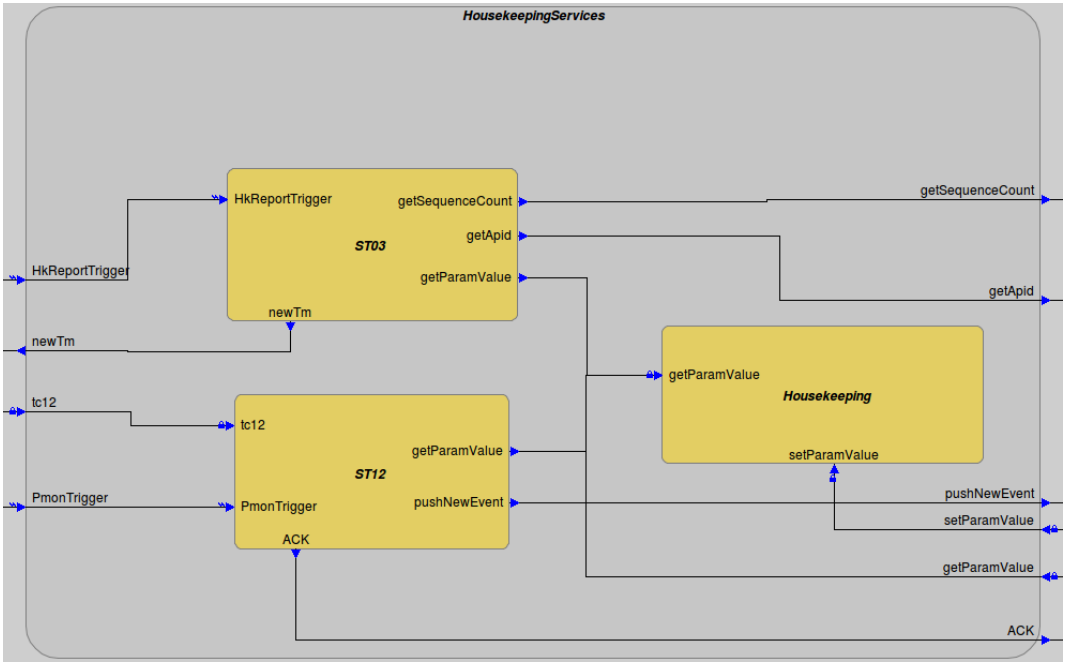


Figure 5-46. Housekeeping Services module

The Parameter Management module includes ST[20] On-board Parameters Management and a function that manage the table of on-board parameters

Provided interfaces:

- tc20: protected, new telecommands for ST[20].
- setOnBoardParam: set the value of an on-board parameter.
- getOnBoardParam: get the value of an on-board parameter.

Required interfaces:

- newTm: telemetry packet to TM queue.
- getSequenceCount: get sequence counter from on-board APID.
- getApid: get APID from on-board APID
- ACK: send the different types of acknowledgement to ST[01] Request Verification.

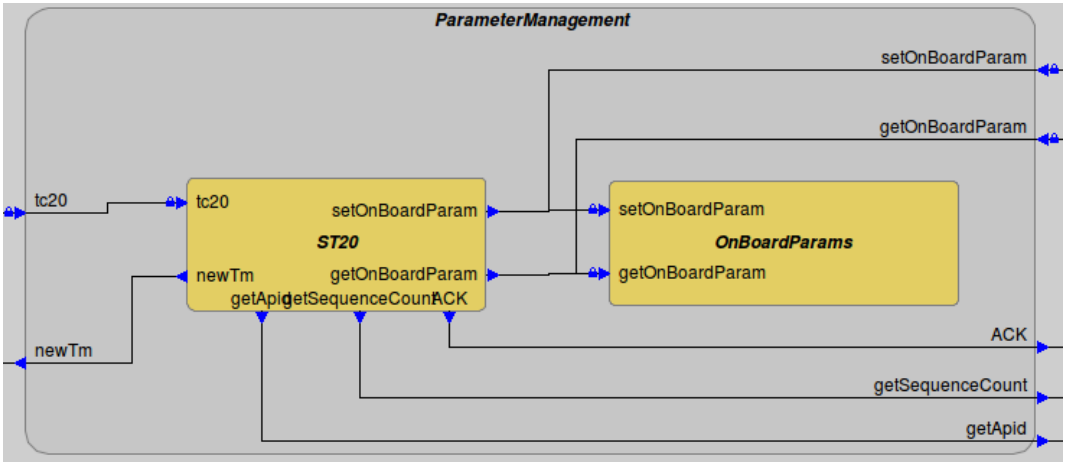


Figure 5-47. Parameter Management module

The File Services service module include ST[12] On-board Monitoring and a function to manage the files with all the functionalities defined in the static part of the PUS services library. There are several interfaces between these two parts, one each function that can be performed.

Provided interface:

- tc23: protected, new telecommands for ST[23].

Required interfaces:

- newTm: telemetry packet to TM queue.
- getSequenceCount: get sequence counter from on-board APID.
- getApid: get APID from on-board APID
- ACK: send the different types of acknowledgement to ST[01] Request Verification.

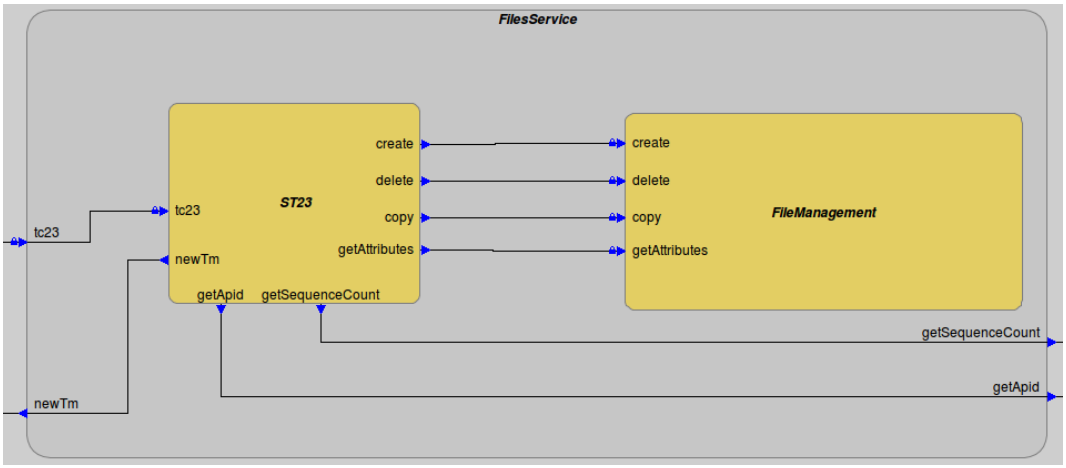


Figure 5-48. File Services module

The OBCP module, as Figure 5-49 shows, is composed by the function to manage the packet creation of ST[18] On-board Control Procedure and by the OBCP Engine that manages all the OBCP procedures.

Provided interface:

- Tc18: protected, new telecommands for ST[18] On-board control procedures.

Required interfaces:

- ACK: send the different types of acknowledgement to ST[01] Request Verification.

- newTc: telecommands packet to Tc queue.
- pushNewEvent:
- setOnBoardParam: set the value of an on-board parameter.
- getOnBoardParam: get the value of an on-board parameter.

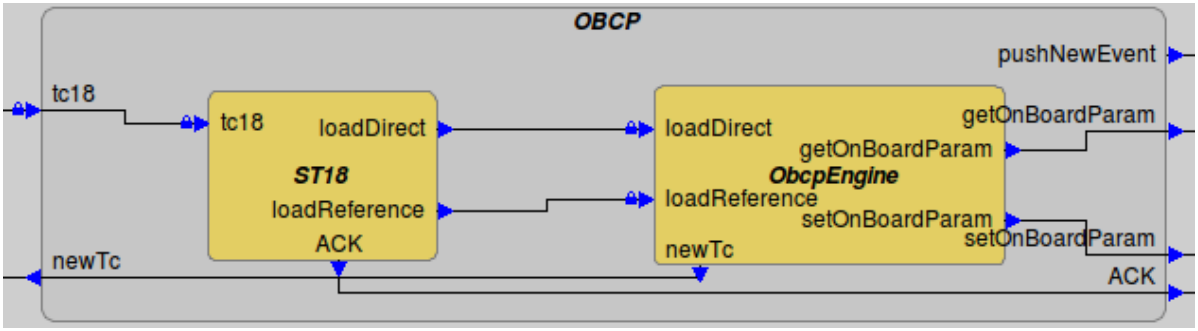


Figure 5-49. OBCP module

In Figure 5-50, several services or functions that perform individually are represented. They do not need a table or a management of the service where others services can access to it.

- ST[01] Request Verification:
  - PI: ACK
  - RI: newTm, getApid, getSequenceCount
- ST[08] Function management:
  - PI: tc08
  - RI: ACK
- ST[11] Time-based scheduling:
  - PI: tc11
  - RI: newTc, ACK
- ST[17] Test Connection:
  - PI: tc17
  - RI: newTm, ACK, getApid, getSequenceCount.
- OnBoardApid
  - PI: getApid, getSequenceCount

All of them have some interfaces in common like getApid, getSequeceCount, ACK and newTm that has been explained above. There are others like tc08, tc11 and tc17 that are used to receive new telecommands.

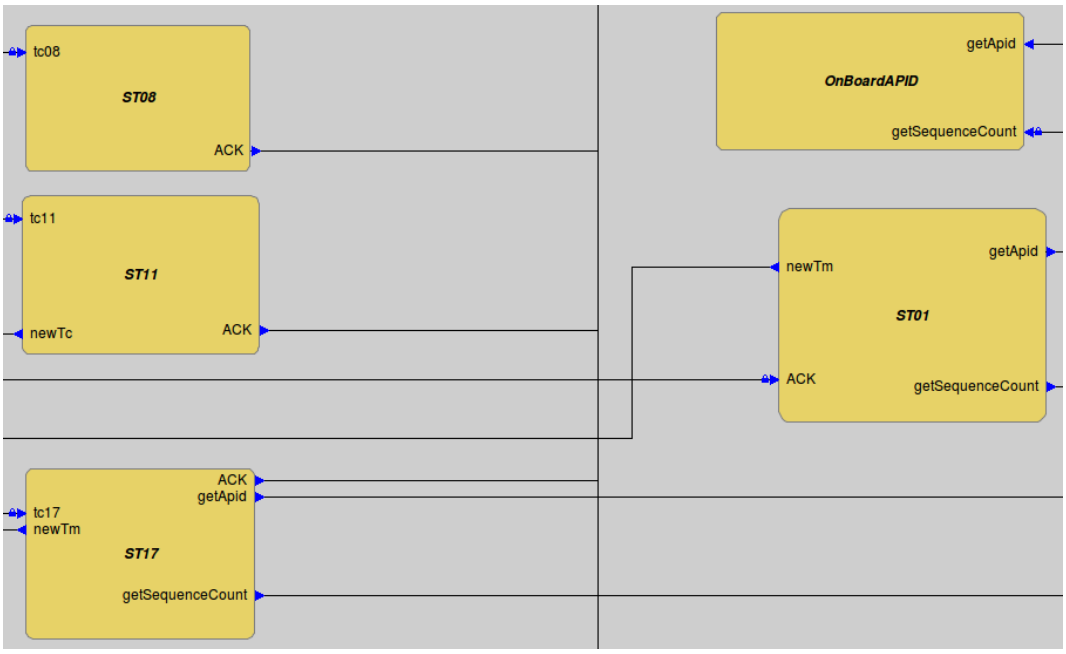


Figure 5-50. Other PUS services

The Figure 5-51 represents functionality of the on-board software. There are some interfaces represented that is connected to the PUS Services library to set or get values to parameters, push new events in the events table of the ST[05] Event Reporting service.

There is also a debugTrigger that is used, in this implementation, to activate different functions to test that everything works in the right way. More interfaces could be added in the future, to manage other functions of the library, for example, send new telecommands, use functions from ST[08] Function management, etc.

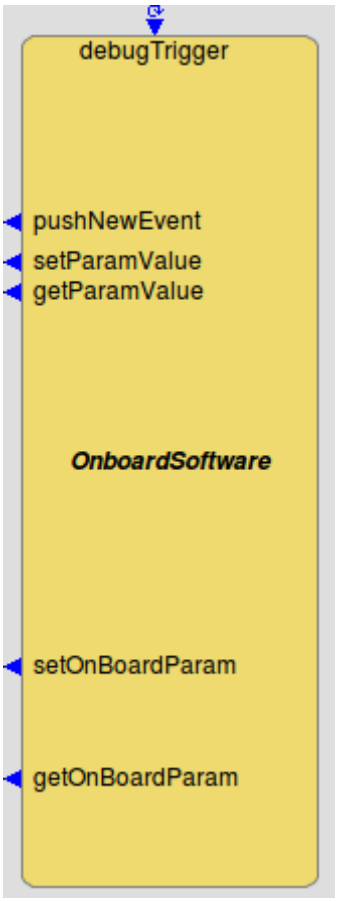


Figure 5-51. PUS services interfaces with the on-board software

5.4. DEPLOYMENT AND EXECUTION OF APPLICATIONS

5.4.1.AIR HYPERVISOR AND HAIR EMULATOR

5.4.1.1. INTRODUCTION

This chapter presents the architecture and design of both AIR hypervisor and its emulator HAIR.

Nevertheless, it is important first to contextualize what are both AIR and HAIR within ESROCOS system, their original architecture and the objectives of the new architecture.

In ESROCOS AIR and HAIR intend to add the mixed criticality capabilities to ESROCOS, meaning an additional software layer between the application and hardware that ensures that a robotic application can be partitioned into a set of isolated systems in both in time through an execution schedule and in space through its own memory region.

Therefore both AIR and HAIR are depicted in Figure 4-1 above the Space and Laboratory target and below TASTE that will implement the software application and also be interface point of AIR and HAIR. AIR is aimed at the Space target while HAIR is aimed at the Laboratory level target.

Since HAIR/AIR would not be no longer a seen as standalone element but an hypervisor that must be built to interface with other elements, in this particular case to interface with TASTE, a new architecture was devised moving from a single purpose solution to a modular architecture flexible to multiple needs.

With AIR/HAIR divided in a set of independent modules (presented in the following sections):

- The integration with TASTE is easier and providing more and dedicated connection points to external elements (e.g. TASTE can connect only to module of device driver)
- The validation and qualification process is more simple because it can be applied independently at each module that are smaller and each with specific requirements.
- The validation process is lighter because out of context modules can be excluded.
- AIR is a lighter and more performant application because it can work only with required modules instead of all modules by default (Figure 5-52).
- Device drivers are modules, allowing to be implementation independent and have been imported from or common to TASTE, RTEMS or other.

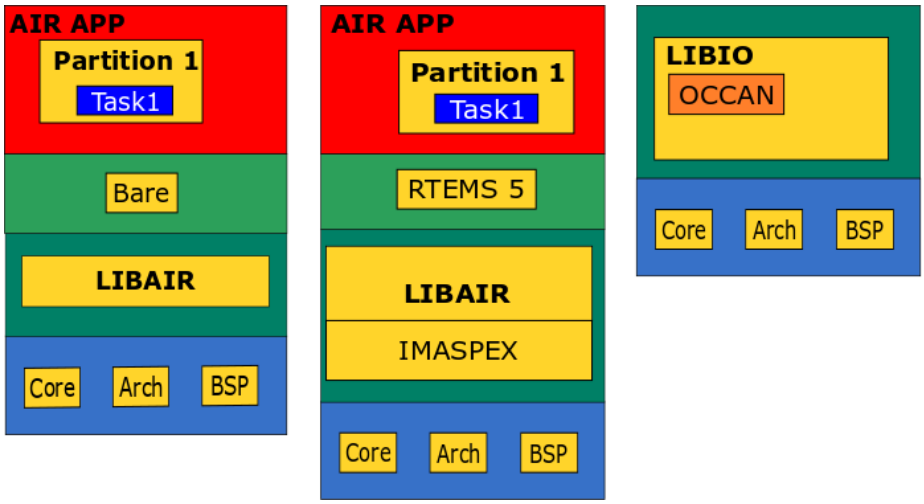


Figure 5-52. AIR/HAIR partition containing only the required modules

5.4.1.2. ARCHITECTURE

The AIR hypervisor architecture's is depicted in Figure 5-53 and consists on the following modules:

- **Partition Management Kernel (PMK):** It holds the main functionality of the hypervisor and implements the Time and Space Partitioning (TSP). In turn it can be divided in three modules:
  - **Core:** which holds the **core functionality** of AIR such as partition management, TSP paradigm, multi-core handling, syscall implementation and health monitor;
  - **Arch:** The generic functionality needed for AIR to run a processor **arch**itecture, currently is only supporting the SPARC architecture;
  - **BSP:** The specific functionality needed for AIR to run a **B**oard **S**upport **P**ackage, which currently supports the LEON2, LEON3 and LEON4 processor boards.
- **Partition Operating System (POS)**: It holds the RTOS being hypervirtualized by AIR, currently the module supports **RTEMS 5**, Qualified **RTEMS 4.8** Improved and **Bare** bone execution without a RTOS.
- **Libs:** This module is composed by a set of libraries implementing functions to bridge the user application, RTOS and PMK. The libraries are:

- **IMASPEX:** Implementation of standard ARINC 653 API which hide the **Core** sub module API holding the system calls.
- **LIBAIR:** Implementation and interface of system calls used by a RTOS trap handler in order to hypervirtualize the RTOS.
- **LIBIOP:** Implementation of device drivers, it re-uses the same device drivers source code present both in TASTE and RTEMS.
- **LIBPRINTF:** Implementation of the *printf* functionality, useful for debugging and pure Embedded RTOS that do not have support to print on a console or other device.
- **LIBTEST:** A set of auxiliary function used to execute, unit, integration and validation test used to be applied in the test and validation campaign of AIR and the device drivers.

- **SW Interference Model (SLIM):** in charge of modelling the effect of SW running in other partitions at partition level. In **Source Code Mode** subset of **SLIM** is used. To avoid an incorrect interpretation this subset shall be called the Partition Interference Model (**PIM**).
- **Tools:** The tools are implemented in parallel to modules presented before, they configure and stich together the previous module to produce the binaries. They divide in two submodules.
  - **Configure**: It configures AIR for a specific architecture, bps and processor. Generating accordingly a set of makefiles for the RTOS and libs modules. It is also used at application level to generate the application's makefile.
  - **Partition Assembler**: It is used at user application level being responsible for stitching the RTOS from POS to everything else according to the user's configuration in terms of schedule and ARINC 653 communication configuration. The partition assembler is capable of setting up the application based on input provided **TASTE**. That is, an intermediate program using as input the **IV, DV and CV of TASTE** information can automatically create the ARINC653 configuration and location of source code skeletons to be used (copied) by the partitions.

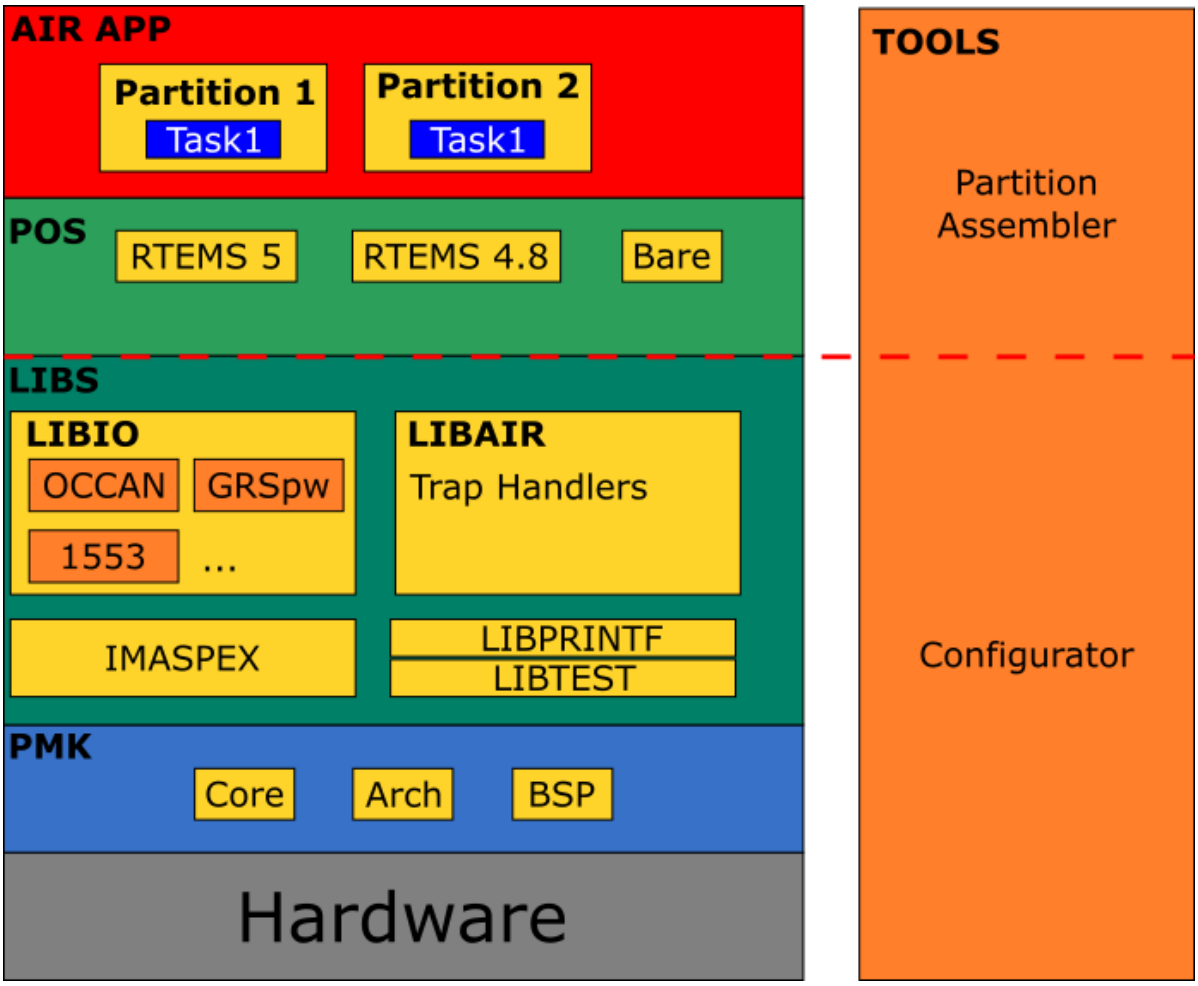


Figure 5-53. AIR Architecture Component Breakdown

Figure 5-54 presents the usage workflow when creating as AIR application to be deployed on an embedded target. Two workflow lines are presented:

- The first workflow line, managed by the **Configure** tool, sets up the build environment for AIR modules and RTOS, thereby generating a set of makefiles. The outcome upon compiling and building are the binaries of the **Libs** and the binaries **install** of the RTOS already hypervirtualized by AIR. On Figure 5-53 these would correspond to the elements below the red dashed line.
- The second workflow line, managed by the **Partition Assembler,** corresponding to the part above the red dashed line, sets up the build environment for an application running on AIR with a paravirtualized RTOS. It creates the configured partition applications, setting them within the respective RTOS **install** and interfacing with AIR system calls through the **Libs**. It can also be seen on Figure 5-54 **that enters the automatic output generated by TASTE**, instead of the user manually create the ARINC 653 configuration, TASTE generates it based on information given and IV, DV and CV.

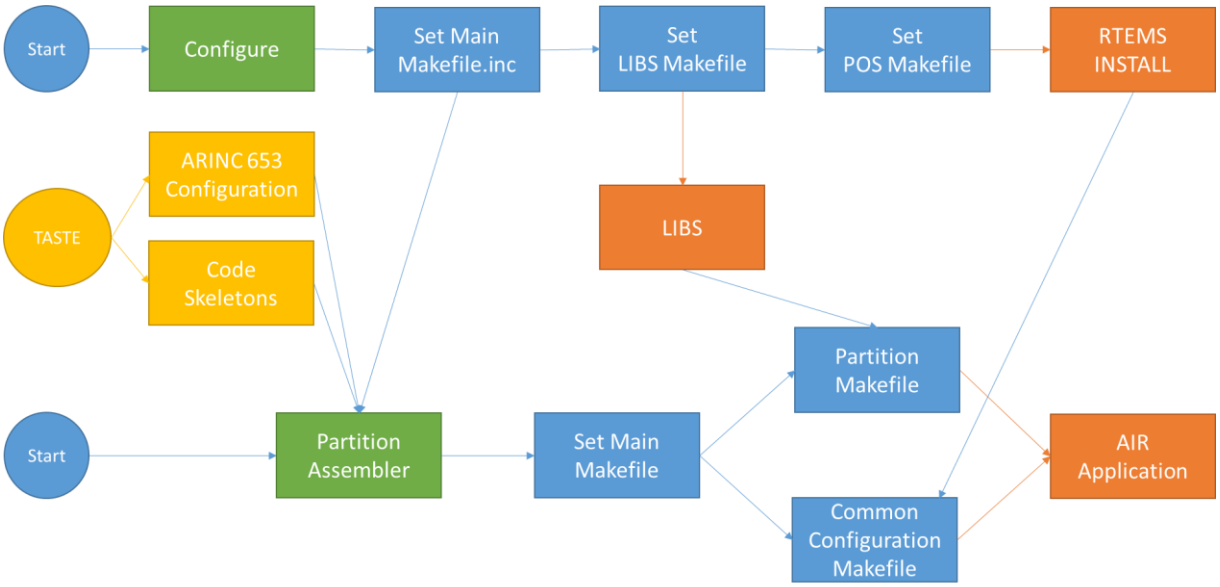


Figure 5-54. AIR Component Workflow

5.4.1.3. SETUP AND INSTALLATION

Concerning the AIR hypervisor, the software is distributed as git repository that can be publicly access and cloned including a script that will perform the installation. Nevertheless, to better understand the software the steps taken by this script in order to install AIR will be detailed.

After cloning AIR's repository the Partitioned Operative System (POS) has to be installed under the POS folder, on AIR's root. Initially only RTEMS 4.8 was supported, however under the purview of this project RTEMS 5 was also integrated as a POS option. The installation of both POS have to follow the same instruction for the normal installation of RTEMS, but the environment variable PATH should be set as the RTEMS bin folder. Furthermore, there is the need to install RTEMS tools, such as the GCC compiler among others. To this end RTEMS website instruction should be followed as well.

To set AIR ready to compile only AIR's **configure tool** has to be run on ARI's root folder. The target processor type and model, which in this project will be SPARC and Leon4, should be choose on the configure tool. Finally the user should run the now generated makefile on AIR's root folder. After this last step AIR is installed.

Besides compiling AIR, for each project under AIR to compile the configure tool and the makefile steps have to be used on the project's root folder.

5.4.2. HAIR EMULATOR

5.4.2.1. ARCHITECTURE

The HAIR emulator consists on the following modules:

- **Manager**: responsible for the management of the hypervisor configuration and simulation time as well as providing the Hypervisor Emulator external interface (including SMP2/SSRA compliance).
- **Hypervisor Partition Emulator (HPE)**: responsible for the system calls emulations in binary mode, a subset of the **HPE** is used in the source code mode being called **Functional Hypervisor Partition Emulator (FHPE)**

- **SW Interference Model (SLIM):** in charge of modelling the effect of SW running in other partitions at partition level. In **Source Code Mode** subset of **SLIM** is used. To avoid an incorrect interpretation this subset shall be called the Partition Interference Model (**PIM**).
- **Processor Emulator (PE) / Simulator API (SAPI)**: it supports the execution of the SW application on the selected host. HAIR is capable of supporting the SPARC IST based ESOC EMU 2.0, (in the case the user is willing to purchase) so the user fully recreates to SPARC processor and uses the same processor instructions as it would use at the target board this is referred as the **Processor Emulator (PE)**. Alternatively, it can run using the host processor in **Source Code Mode** being referred as the **Simulator API**.
- **Hardware Interference Model (HLIM):** Models the influence at HW level of SW running in other cores/partitions. Since the **Source Code Mode** does not emulate the time issues at HW level, this model is not applicable in this mode.
- **Simulator Framework**: provide interfaces for the integration into TSIM and SMP2/SRR2 based frameworks.

For each HAIR module, a component was defined. It will also represent a package containing a set of classes implementing the functionality associated to the components.

The figure below, depicts all components of HAIR, the connection between the components show the existing internal interfaces of HAIR, with exception of the relationship of **FHPE** with **HPE,** meaning the first uses functionality of the second and likewise the relationship between **PIM and SLIM**

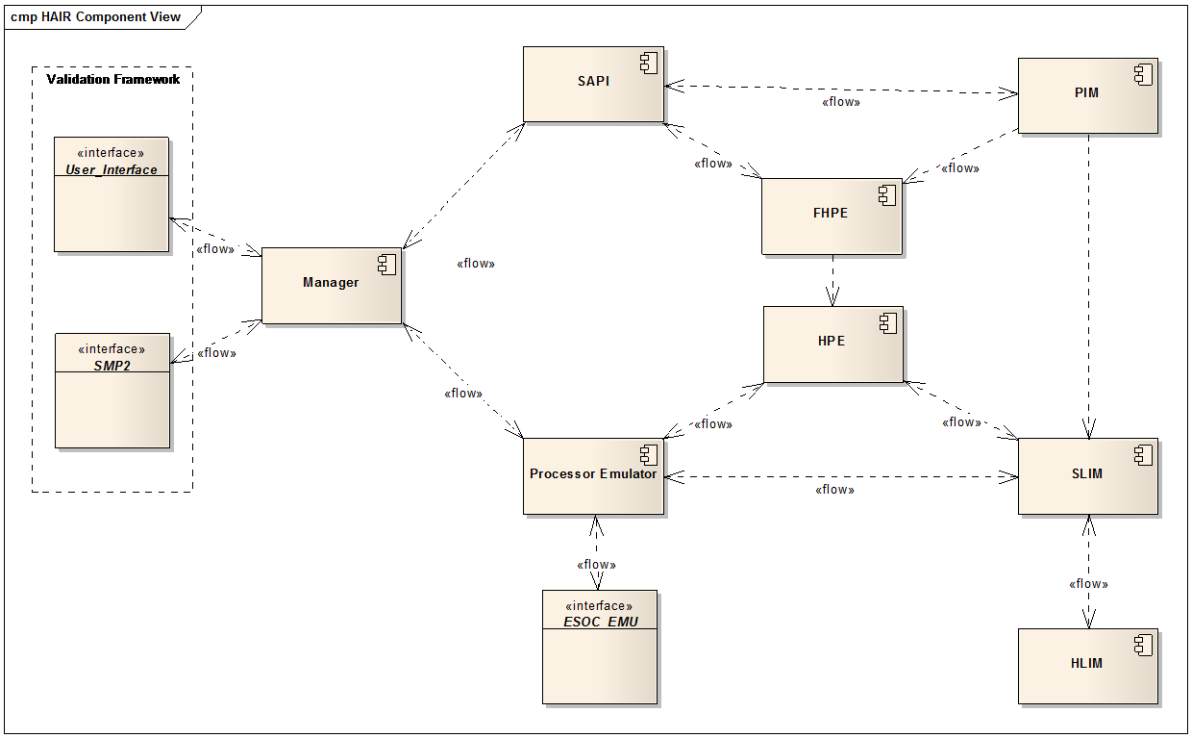


Figure 5-55. HAIR components

The presented components have been divided in two sub sets for the development of HAIR in **Source Code Mode** and another for **Binary Mode**.

The **Manager** is the only component common to both modes presenting the same external interfaces and containing the logic to switch between modes as sub modes as presented in the state machine of Figure 5-56.

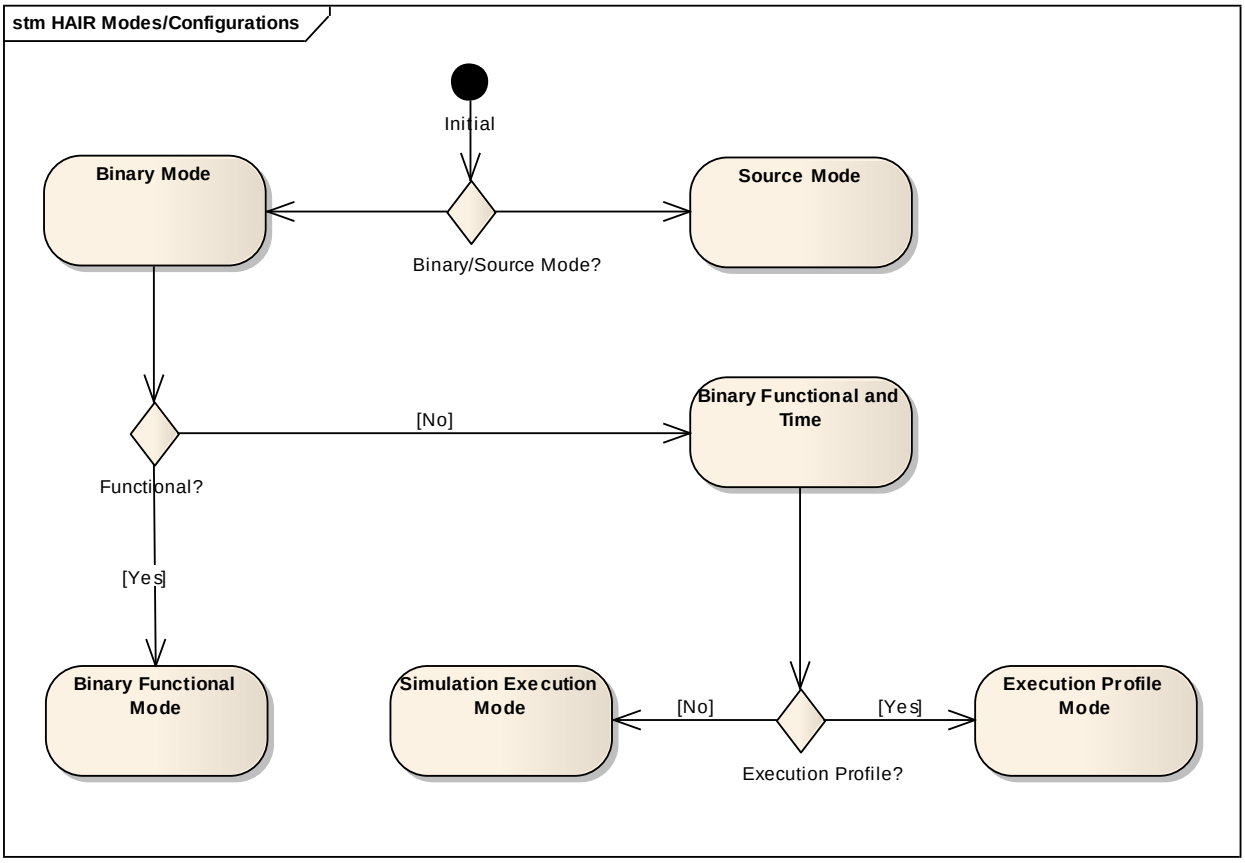


Figure 5-56. HAIR Modes and Configurations

The components are developed inside a set of libraries represented as the design packages. The packages have been organized according with the responsible developer and its functionality. The HAIR libraries/packages are depicted below containing its main classes.

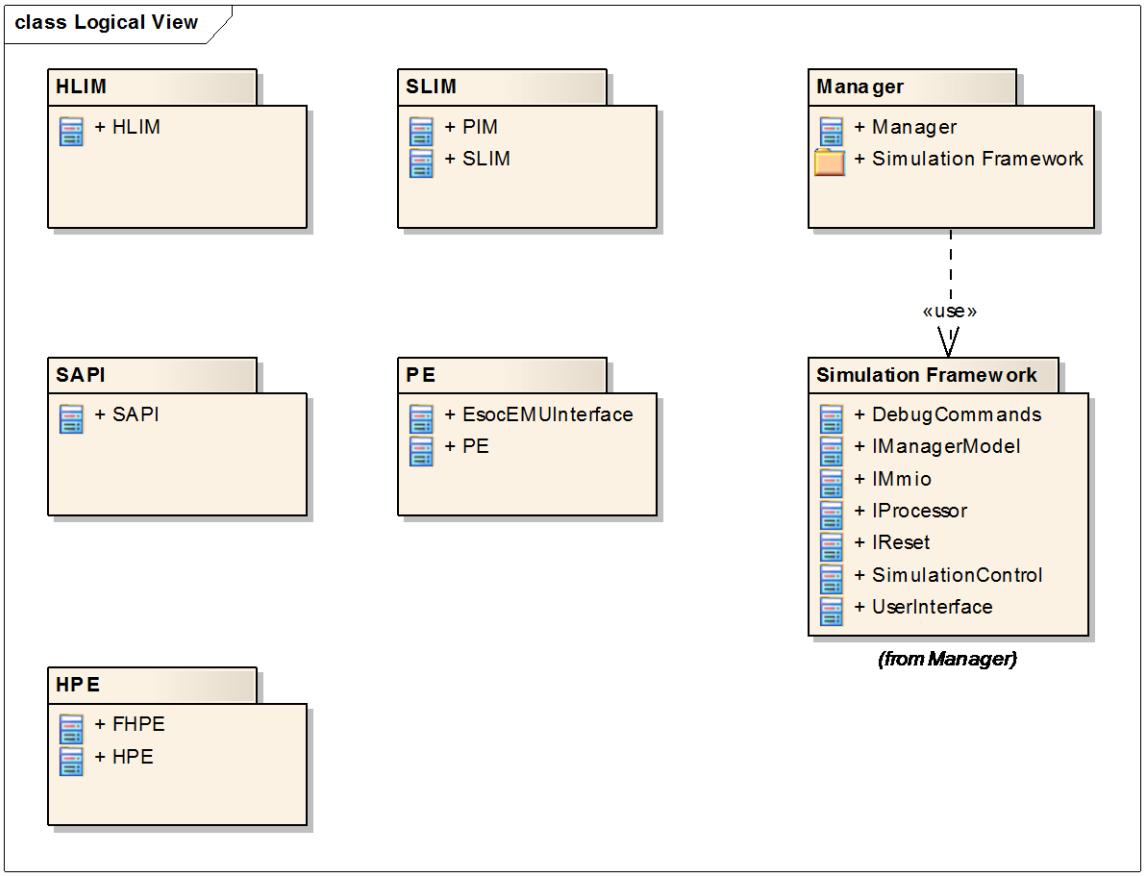


Figure 5-57. HAIR Packages/Libraries

5.4.2.2. SETUP AND INSTALLATION

An installation package is provided with all required binaries, the respective user manual instructs the package installation and required libraries of the host operating system, in fact the user manual provides full details on everything regarding installation and setup. It is specific for different linux distributions, provide all type of examples, all supported commands and configuration options. Here it is only given a generic description of the nominal installation and setup.

The HAIR environment is ready to run transparently AIR applications, it is only required to set an environment folder, where inside the folder a link copy to respective AIR environment folder is created.

The next it is needed to set of parallel input and output folders, the input folder contains a set of simulation control features (run, start, stop, debug elements…) in a **HAIR configuration file** and while the output is gather all results of the simulation.

The last step is to set ready the HAIR environment by executing "***hairdresser*** *<AIR configuration file>"* and then compile the partitions by running "make". With binaries created, the execution of "hair", will automatically simulate the application in the host machine.

5.4.3. DEVICE DRIVERS

The communication with external devices via some typical data links used in space systems and robotics has been identified as a relevant capability for ESROCOS. For this reason, the framework provides a set of device drivers targeted to the GR740 board, the reference avionics platform for the ESROCOS project.

As part of the ESROCOS activities, drivers for the GRCAN controller for CAN bus, the GRSPW2 controller for SpaceWire, and GRETH driver for Ethernet have been integrated into the version of RTEMS used by ESROCOS (4.12). In addition, a port of the Simple Open EtherCAT Master (SOEM) library that implements an EtherCAT master node has been integrated in RTEMS. The drivers are also made available at the level of the AIR hypervisor.

5.4.3.1. CAN BUS DRIVER

In order for TASTE generate code able to communicate using a CAN bus, the PolyORB-HI-C distribution is extended with a new CANBUS driver (po-hi-candriver class).

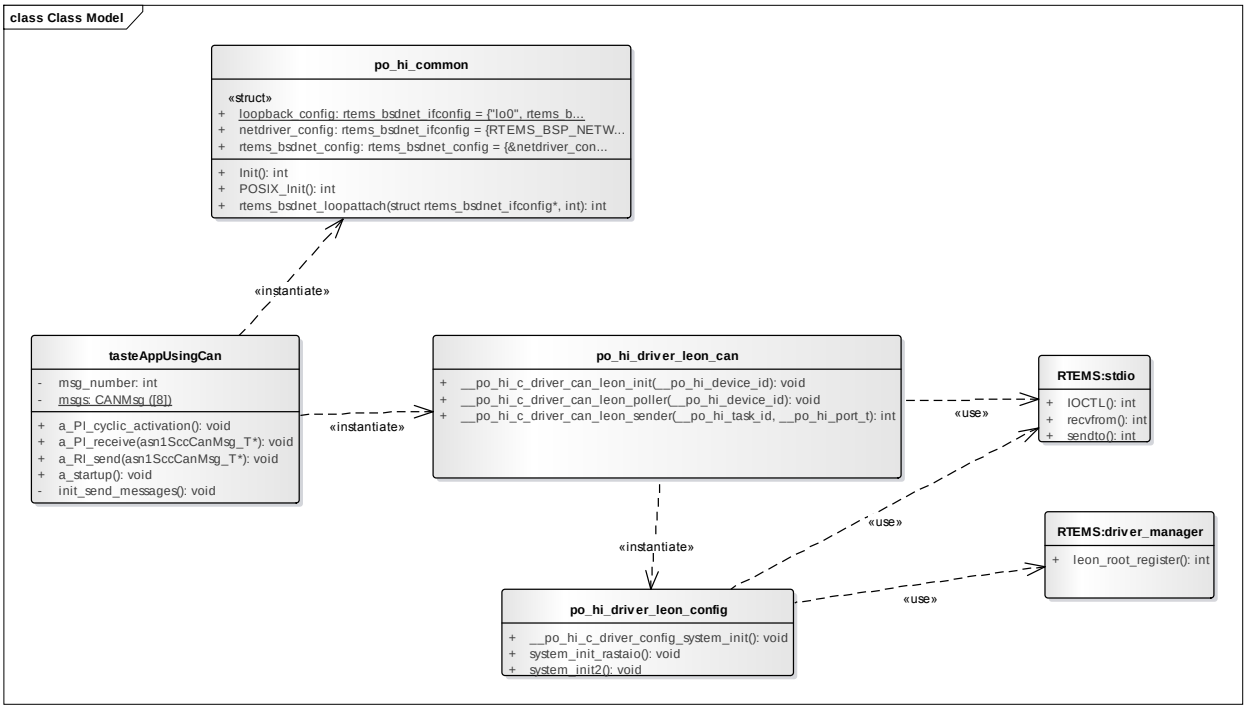


Figure 5-58. Class diagram for the PolyORB-HI CAN driver

The class diagram features the functionality provided by RTEMS through is DriverManager and also typical standard IO API for open/close/read/write from through a file descriptor.

The TASTEAppUsingCan class features the functionality implemented in TASTE to use the drivers, the figures below sample a typical TASTE interface view specification being transparent to CAN usage. It also depicts a TASTE deployment view where it is explicit the CAN driver and respective bus.

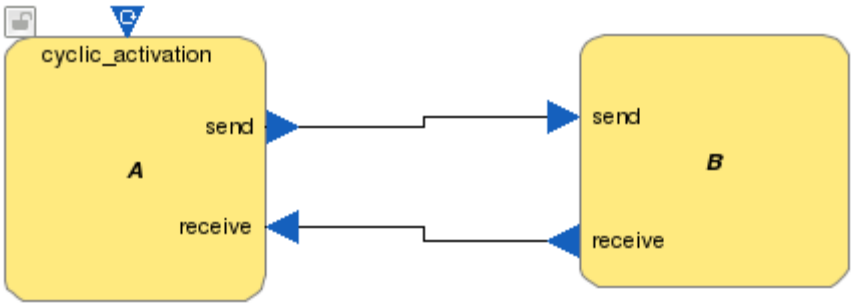


Figure 5-59. Typical interface view used for CAN driver

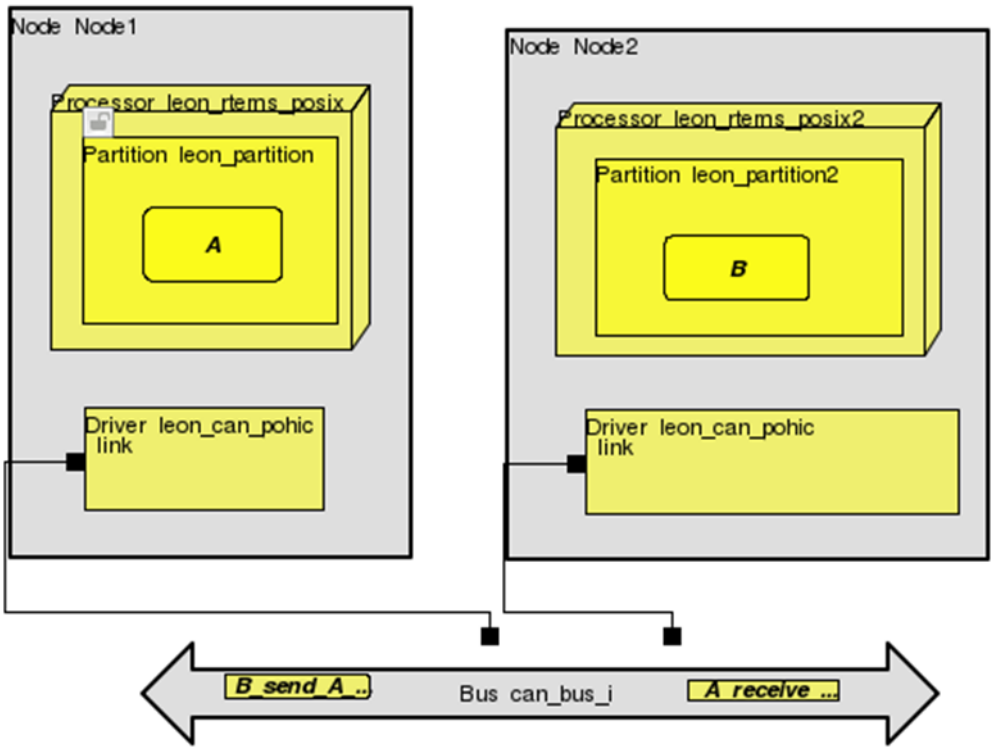


Figure 5-60. Deployment view specifying CAN driver and bus

The provided functionality of the CANBUS driver follows the already used "initialization, poller and sender" philosophy of the remaining PolyORB-HI-C drivers. Therefore the driver's main functions are:

- po_hi_c_driver_can_leon_init – It is the starting point in using an Ethernet device for communication.
- po_hi_c_driver_can_leon_poller – A continuous loop waits for a CAN data until receiving using the stdio read primitive.
- po_hi_c_driver_can_leon_sender – Sends data to a CAN device through its associated stdio write primitive.

Since CAN bus works as peer to peer communication being all configuration explicit at TASTE deployment view, the driver does not hold itself any specific data structure.

Regarding the integration of the device driver in AIR hypervisor, the functionality above is copied into AIR's LIBIOP (see 5.4.1), where AIR implements a middle layer to interface with detailed (in D3.4 document). By maintaining a copy of the device driver functionality, it is ensured to apply the same validation process on the device drivers in both AIR and without AIR environments.

5.4.3.2. ETHERNET/ETHERCAT DRIVER

An EtherCAT master implementation is provided by the Open EtherCAT Society as the Simple Open EtherCAT Master (SOEM). The SOEM can be used to implement an EtherCAT network and only requires access to the physical ethernet device from the operating system.

In the scope of the project the SOEM functionality has been extended to support RTEMS with rtems-libbsd. All these developments have been pushed upstream and are readily available for use.

In the case of TASTE, it must build the SOEM library during installation. After that a component can include the SOEM headers and develop an application. This application should be linked against the SOEM library provided in RTEMS.

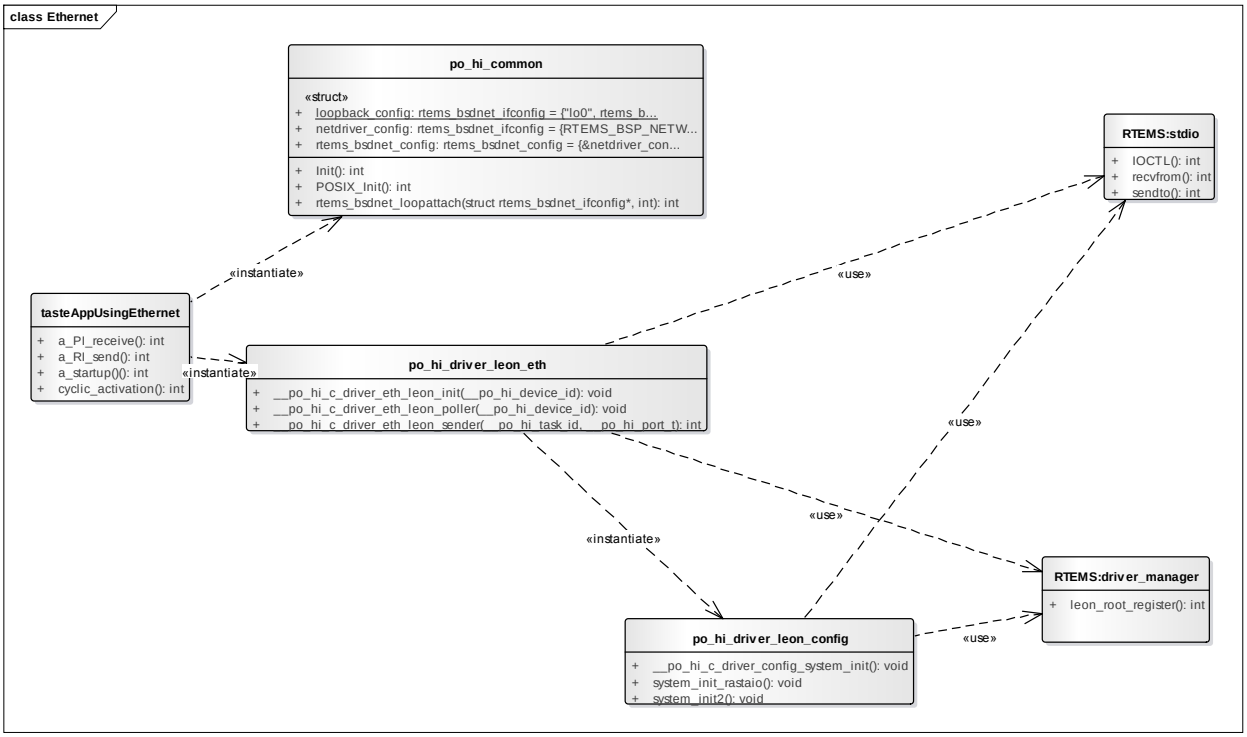


Figure 5-61. Class diagram for the PolyORB-HI low-level Ethernet driver

The PolyORB-HI Ethernet driver is represented in Figure 5-61 with "po_hi_driver_leon_eth" class, the functionality set available by the driver is similar to most PolyORB-HI-C drivers, resuming to the initialization, poller and sender paradigm, where in this case are:

- po_hi_c_driver_eth_leon_init – It is the starting point in using an Ethernet device for communication.
- po_hi_c_driver_eth_leon_poller – Uses the listening socket created at initialization to establish connection to incoming devices sending data, creating a specific socket for it.
- po_hi_c_driver_serial_eth_sender – Sends data to a device through its associated socket.

The main data structures of the driver are:

- **nodes** – Data structure holding a list of sockets connected to other devices used for the transmission of data.
- **rnodes** – Data structure holding a list of sockets connected to other devices used for the reception of data.
- **leon_eth_device_id** – Identifier of the Ethernet device mapped to TASTE component interface.

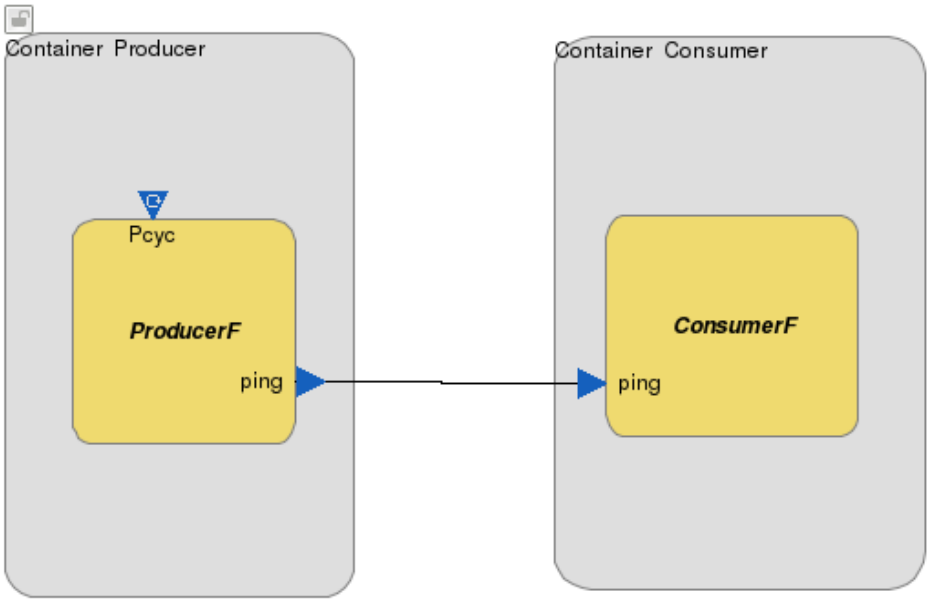


Figure 5-62. Typical TASTE Interface View used for Ethernet driver

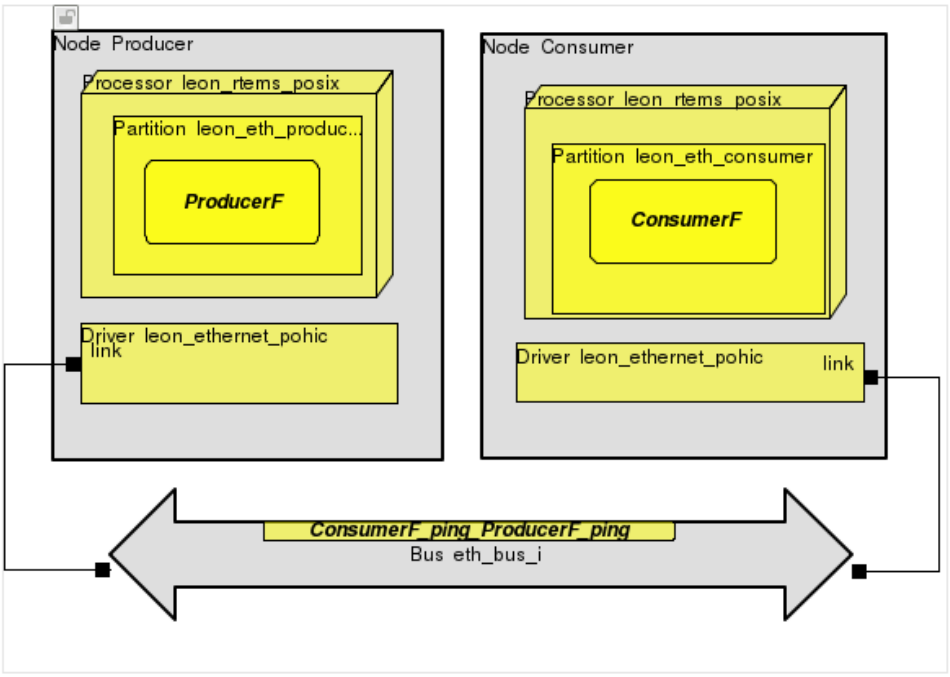


Figure 5-63. TASTE Deployment View specifying Ethernet driver and bus

Regarding the integration of the device driver in AIR hypervisor, please refer to section 5.4.1.2.

5.4.3.3. SPACEWIRE DRIVER

Although the PolyORB-HI-C distribution already includes a SpaceWire driver, a new version is developed for ESROCOS using the new RTEMS DriverManager and Spacewire router libraries.

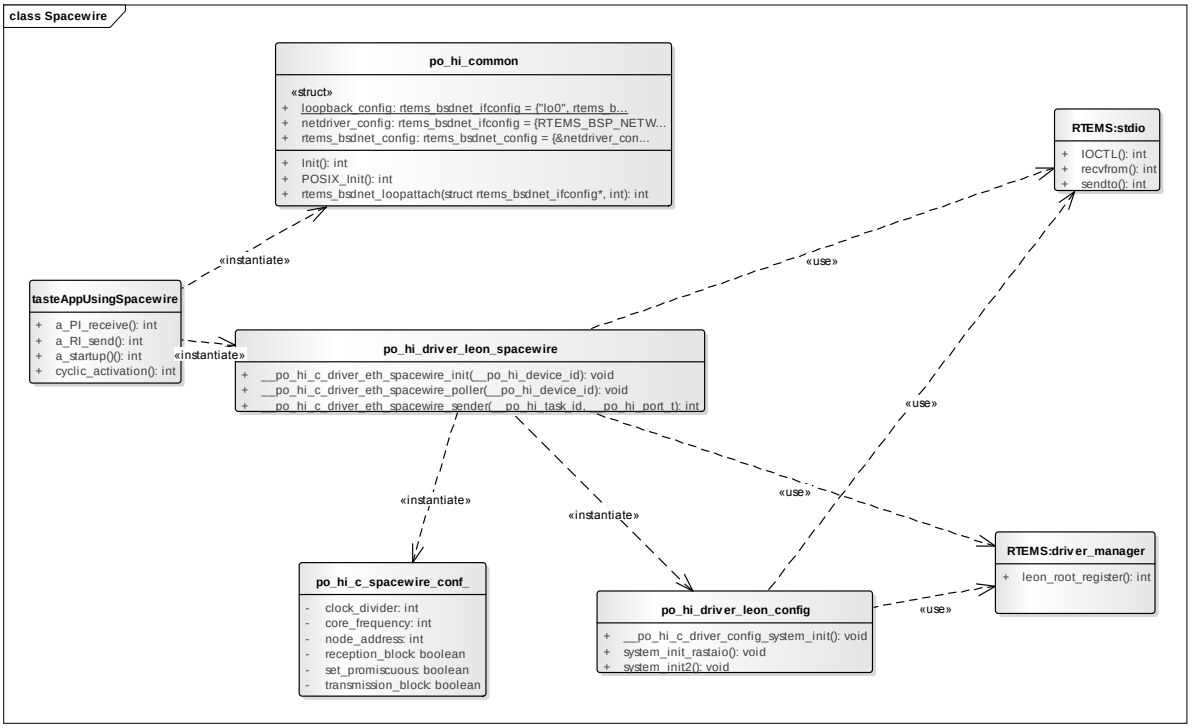


Figure 5-64. Class diagram for the PolyORB-HI low-level SpaceWire driver

The PolyORB-HI SpaceWire driver is represented in Figure 5-64 with "po_hi_driver_leon_Spacewire" class, the functionality set available by the driver is similar to most PolyORB-HI-C drivers, resuming to the initialization, poller and sender paradigm, where in this case are:

- po_hi_c_driver_eth_Spacewire_init – It is the starting point in using an SpaceWire device for communication, it also enables the SpaceWire initialization protocol to plug and play any node.
- po_hi_c_driver_eth_Spacewire_poller – Uses the listening SpaceWire node created at initialization to establish connection to incoming routed nodes sending data.
- po_hi_c_driver_serial_Spacewire_sender – Sends data to a SpaceWire terminal through its associated routed node.

The main data structures of the driver are:

- **__po_hi_c_Spacewire_conf_t** – Data structure holding the configuration of a Spacewire, it includes the following information:
  - Node address
  - Core frequency or Clock Divider value
  - Transmission and Reception blocking behaviour
  - Promiscuous Behaviour

- **fd** – Data structure holding a list of files descriptors associated to the created Spacewire nodes, these are the unique ids reference to where to send and transmit data.
- **leon_Spacewire_device_id** – Identifier of the SpaceWire device mapped to TASTE component interface.

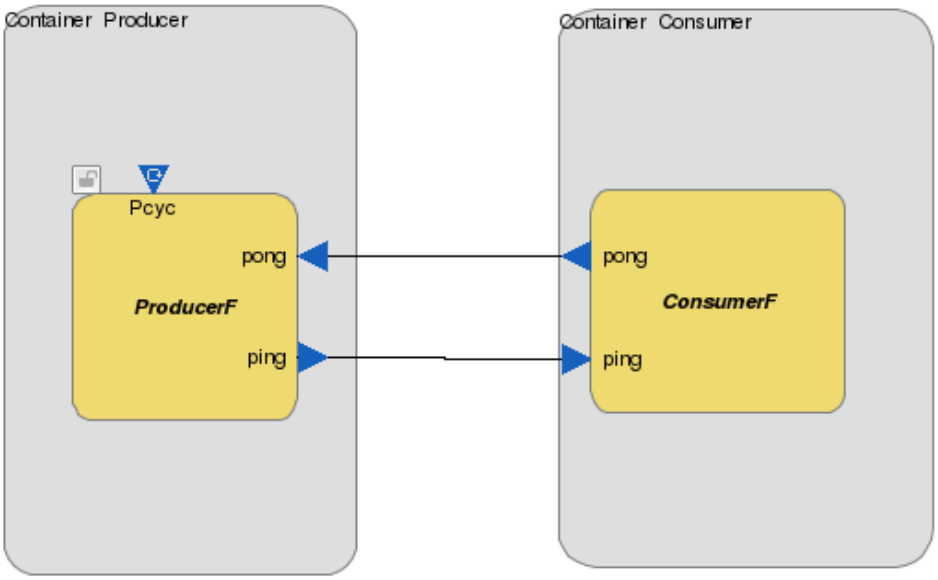


Figure 5-65. Typical TASTE Interface View used for SpaceWire driver

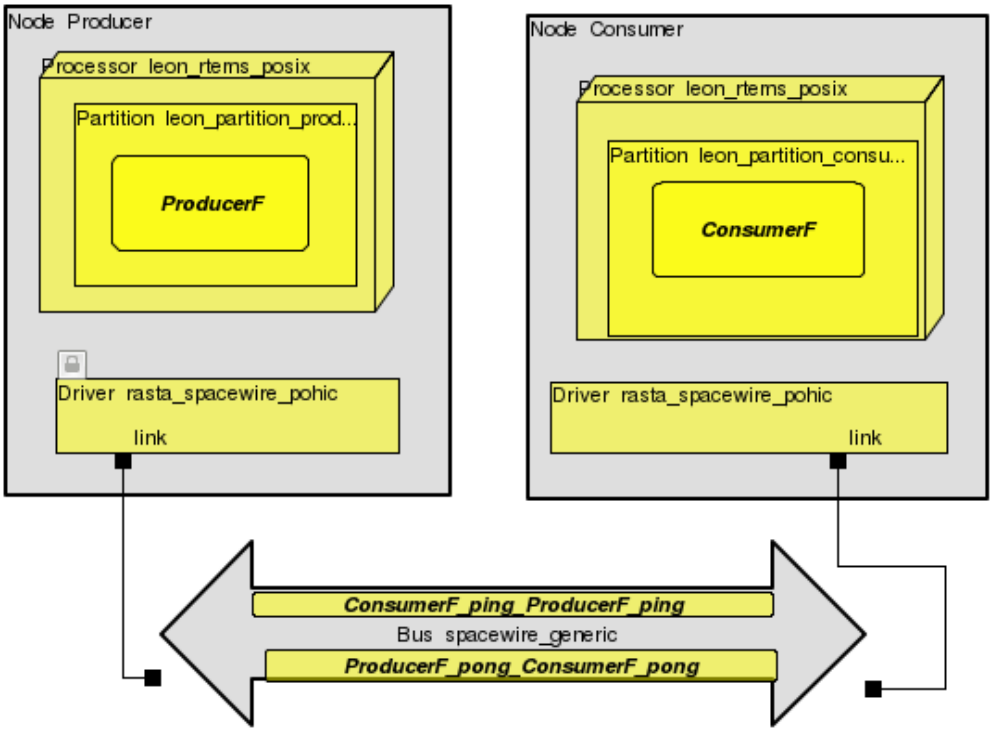


Figure 5-66. TASTE Deployment View specifying SpaceWire driver and bus

5.5. MONITORING, DEBUGGING AND TESTING

5.5.1. DATA LOGGER

The logging and replay of timestamped data is essential during the development of robotics applications. These applications are complex and dynamic, so the capability to store test data and replay is invaluable in performing repeatable tests and investigating issues. To this end, logging is often provided at the level of the middleware interfaces of the application, so that the inputs and outputs of individual software components can be stored, analysed and reproduced.

The data logger functionality can be separated into two use cases:

- *Capture*: Recording of log files from data that is transmitted between TASTE components in a robot control application
- *Replay*: Inserting data from log files into a robot control application

To capture log files, it is required to intercept data that is exchanged with RI-PI connections from a TASTE application and store it to a mass storage. To achieve this functionality, the following building blocks will be required:

- A mechanism to buffer data generated within a TASTE application
- An API to access and fetch the buffered data
- A library for performant writing and reading of log data

In addition, the support of a replay mechanism will require adjustments in the TASTE PolyORB-HI/C middleware so as to support proper event injection.

This component is at the intersection of multiple concerns: the data logger functionality will help:

- Supporting a required feature that would ease testing for the robotics domain;
- Implementing a functionality that is similar to some PUS services also discussed with the TASTE design team at ESA;

Logging and replay will also interfere with the regular flow of execution of the system It is therefore required to evaluate its impact on performance metrics such as schedulability and memory. At first, we target a lab quality implementation, with provision to support part of these functions in SPACE quality setting.

5.5.1.1. DATA LOGGER GENERAL ARCHITECTURE

In the section, we elicit the high-level elements of the data logging service.

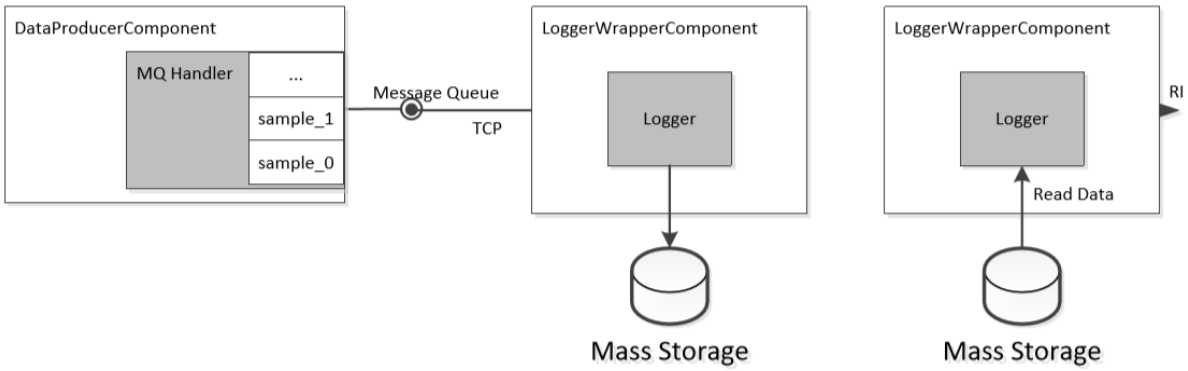


Figure 5-67. Logger Architecture for log file recording (left) and replay (right). Grey boxes are libraries, white boxes TASTE components

In Figure 5-67 we show on the left the architecture of a TASTE program with a logger attached to a data stream. The following individual items can be identified in the drawing:

- Data Producer Component: this stands for any component in a TASTE application, which produces the data stream which should be logged. This could for example be a sensor or data processing algorithm.
- MQ Handler: At runtime, the Logger must intercept data that is exchanged with RI-PI connections from a TASTE application. To get access to the data, a new TASTE feature is being developed allowing to make available a queue of data samples for remote access. The MQ Handler is added to a component via code generation. It provides a growing queue of data samples, and an API which gives remote access the queue via TCP. This MQ handler is an extension of the existing MQ handler present in the TASTE middleware.

  The original design only contemplate circulating the message among components. Extension will consider the following additional capabilities:
- Persistent storage: currently, consumed messages are destroyed. In order to preserve efficiency, the internal design will revisit the lifecycle of messages to reduce message copying and simply move messages from "to-be-consumed" queue to a "logged-messages" queue;
- Timestamping mechanism: the current implementation does not timestamp messages. These are implicitly timestamped by the time of consumption. This must be adjusted based on semantics of the logger: timestamping at production time and at consumption time.
- Logger: The main logging functionality will be made available as C++ library. The library provides an API for manipulating log files and for writing samples and reading samples to and from log files.
- Mass Storage: The logger will required access to a mass storage such as a SSD or HDD as target device for the log file. This API will be made available as a set of library elements, an initial design is present in `po_hi_storage.h` from the PolyORB-HI/C repository.
- LoggerWrapperComponent: In order to make available the logger within a TASTE application, a user-written wrapper component is required, which calls both, the Message Queue API to receive samples and the Logger API to store them. For the case of replaying logs files, the LoggerWrapperComponent calls the logger API for retrieving samples and uses them as arguments in RI calls.

From this set of elements, it appears that the most critical aspects of the design are:

- The updated lifecycle of messages: message must be kept for log inspections. Yet, this must not have a significant impact on the qualification effort that imposes static dimensioning of resources, among other constraints;
- The expected performance of the logger so as to inspect precisely the behaviour of the system. Depending on the inspection points, it can scale from basic information such as a position to a full image as captured by a camera.

Note: DFKI, ISAE, GMV and TASTE ESA team held various meetings to clarify the design intent and discuss the options towards implementation. We summarize this discussion in the next sections.

5.5.1.2. MESSAGE QUEUE BUFFERING DESIGN STRATEGY

Implementing the logging strategy imposes to revisit the message queue buffering strategy currently in place. For the moment, the design statically allocates one message queue per generated thread. The size of this message queue is statically configured by the designer.

In order to implement the logging mechanisms, and reduce the induced overhead, the following design choices have been made:

- Inspection points are statically known at design time, along with key configuration parameters: number of messages to be captured per unit of time (all, one every n messages), the global number of messages to be kept, and the strategy in case of overflow (keep latest, oldest, error).
- Logged information is the message exchanged by the middleware itself. A post-processing of this message using ASN.1 marshallers will be necessary during analysis. Manipulating middleware messages reduce the complexity of the logger logic to simple manipulation of existing data, and does not require additional processing at run-time.

Based on these considerations, the logging mechanism is a variant of the existing internal middleware queue:

For the moment, PolyORB-HI/C implements on global message queue serving all threads. This message queue stores middleware messages, and dispatch them to their consumer. It is already statically allocated, and implement buffer overflow strategies (keep latest, oldest, error). The logger queue can be seen as an additional consumer of specific messages. This design meets the requirements of limited impact on the resources used by the middleware.

In complement, an API will be made available to receive these messages for processing, the logger API presented in the next section.

5.5.1.3. LOGGER LIBRARY

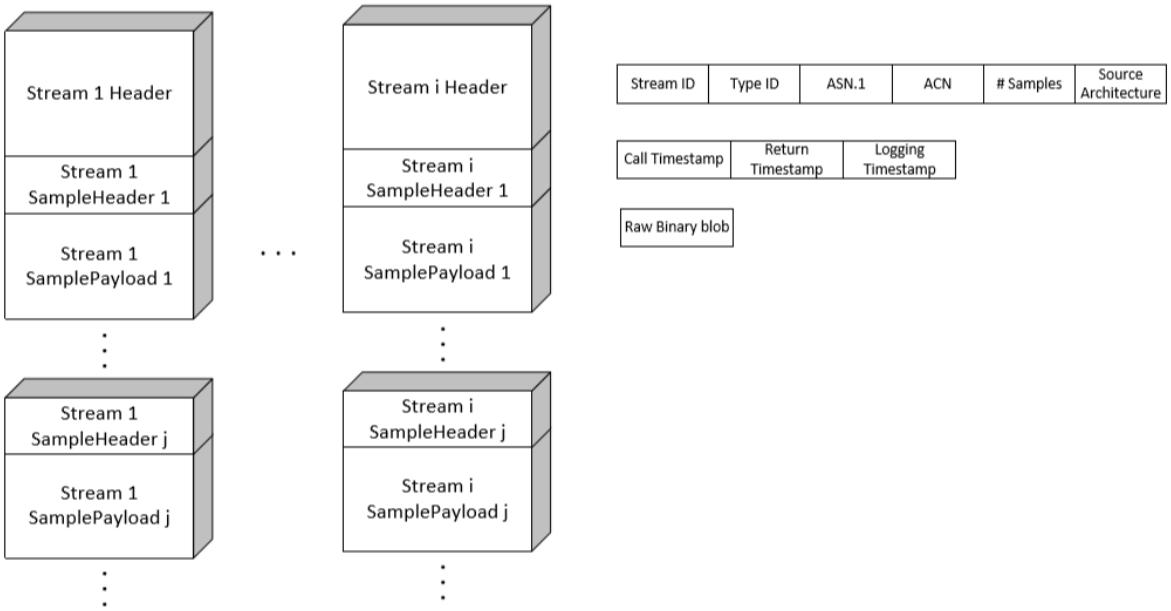


Figure 5-68. Memory layout of a log file

The log file itself is stored in a way that all necessary information for the reconstruction of the data it contains is present in the log file. We show in Figure 5-68 the design of the log files. Per data type used within a logged TASTE interface, we create an individual data stream, which is stored to a separate log file. The payload data of the stream (i.e. the actual data sample) is stored as binary blob, which is serialized with the ASN.1 serialization function defined by the ASN.1 and ACN information stored in the header in the stream. Additionally, for each sample a small header containing timing information will be stored.

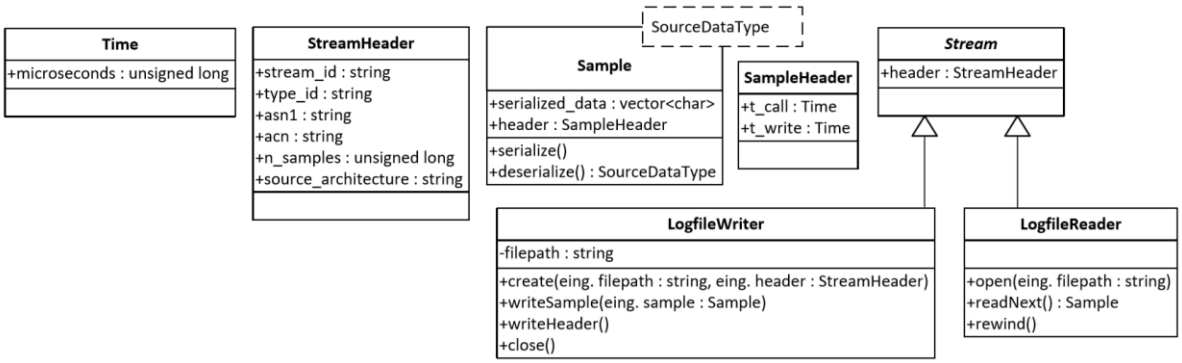


Figure 5-69. Reduced view on the API of the logger library

All interaction with the log files is done using the C++ API of the logger. The diagram shown in Figure 5-69 shows a sketch of the API. The API is dependent on many domain usage that are in discussion both with ESA and internally to the ESROCOS consortium. This API presents basic mechanisms to store samples. The actual API is likely to differ once all usage domains are known. It is the reason the final API is still under construction and will be refined during the implementation and testing phases. In addition, the shown API is reduced in a way, that it only contains public functions and attributes.

5.5.2. VISUALIZATION AND SIMULATION TOOLS

The usage of tools to visualize the robot behaviour and the data from sensors and algorithms is essential in the development of robotics applications, as is the capability to simulate the robot and its environment for testing purposes.

ESROCOS does not aim to develop new visualization or simulation tools, but to integrate existing, mature and high-quality tools into the framework so that they can interface with the applications developed with the framework. These tools belong to the laboratory environment, but they can interface both with applications running on a laboratory or a space quality target.

The visualization tools selected for integration in ESROCOS are vizkit3d (from the ROCK ecosystem) and RViz (from the ROS ecosystem). vizkit3d is oriented to the 3D visualization of robot pose and sensor data, while RViz allows for the visualization of other types of sensor data, including 2D data such as images. The two tools should offer the user a good choice of data representations.

As for simulation, the Gazebo simulator has been chosen for integration in ESROCOS. Gazebo is ready to use with the ROS framework, and it will be integrated in ESROCOS using the generic mechanisms provided for the integration of the TASTE and ROS middleware environments.

5.5.2.1. VIZKIT3D INTEGRATION

The vizkit3d integration in TASTE makes use of several components derived from the SARGON project and adapted to the TASTE and build system versions used in ESROCOS:

- **gui/TASTE/vizkit3d:** a set of TASTE functions corresponding to each vizkit3d plugin that contains a data update function.
- **gui/vizkit3d_TASTE**: a C/C++ library that manages the initialization and data input of the vizkit3d application and provides a C wrapper interface for integration with TASTE.
- **types/base** and **types/sensor_samples:** two sets of ASN.1 types generated derived from ROCK (these types are identical to the ones defined in the dataview-uniq.h file used internally by the TASTE build process); see section 5.3.1.
- **types/base-support** and **types/sensor_samples_support:** libraries that provide conversions between the C++ types used by vizkit3d and the C types used by TASTE, as well as utility functions.

In addition, it relies on the following third-party libraries:

- **gui/vizkit3d:** the vizkit3d visualization application.
- **gui/robot_model:** a vizkit3d plugin for visualizing a robot URDF model.
- **base/types:** the ROCK library of core robotics types (vizkit3d renders data in the base-types formats).
- **LibYAML-cpp:** an open-source YAML parsing library.

Figure 5-70 presents the architecture and the dependencies among these components.

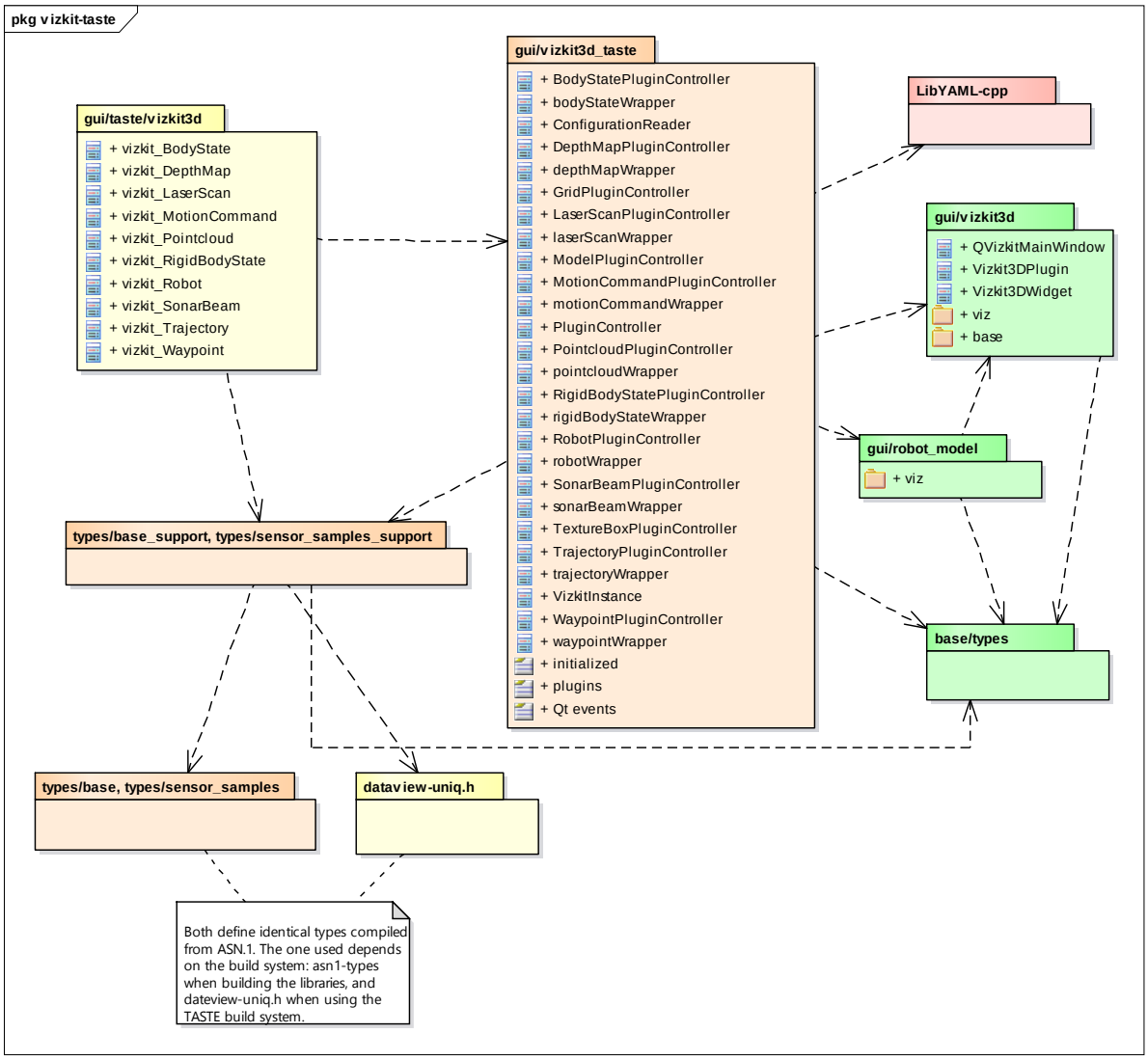


Figure 5-70. Component architecture of the vizkit3d-TASTE integration

The vizkit3d application provides thirteen basic plugins. These are divided in two categories:

- Base plugins (vizkit3d package, namespace 'base'), which render dynamic data of a number of types defined in the base/types library:
  - BodyStateVisualization
  - DepthMapVisualization
  - LaserScanVisualization
  - MotionCommandVisualization
  - PointcloudVisualization
  - RigidBodyStateVisualization
  - SonarBeamVisualization
  - TrajectoryVisualization
  - WaypointVisualization

- Visualization plugins (vizkit3d package, namespace 'viz'), which render static 3D elements that support data visualization:
  - GridVisualization
  - ModelVisualization
  - TextureBoxVisualization
- Robot model plugin (robot_model package, namespace 'viz'), which renders a 3D robot model.
  - RobotVisualization

For each basic plugin a TASTE function is defined, as depicted in Figure 5-71. These functions can be included in a TASTE Interface View, and will create an instance of a vizkit3d plugin of the corresponding type that renders one kind of data.

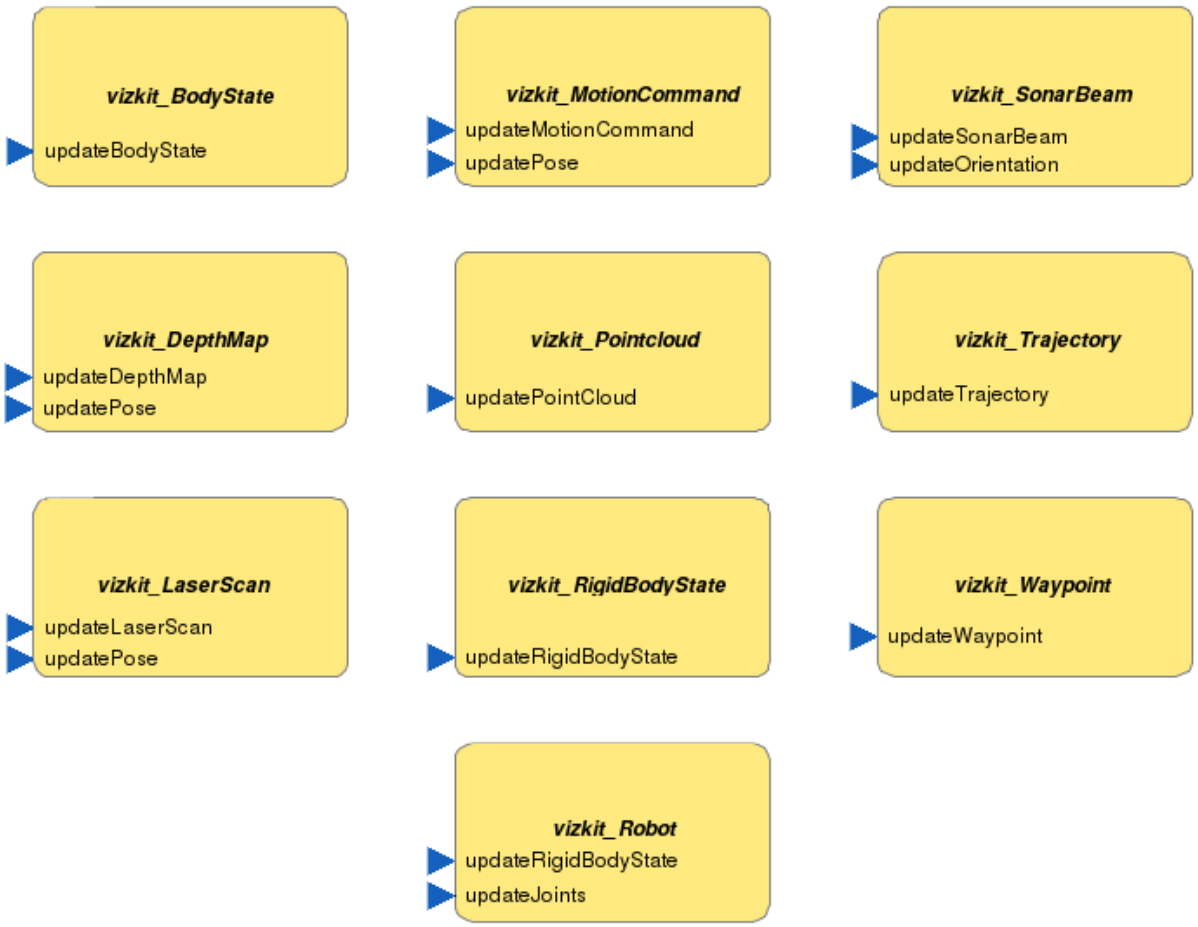


Figure 5-71. vizkit3d plugin functions in TASTE

The architecture of vizkit3d is summarized in Figure 5-72. A vizkit3d application contains a single QVizkitMainWindow object, which is a Qt application window with a 3D widget (class Vizkit3DWidget). Different plugins, defined by the superclass Vizkit3DPlugin, can be added to the main window, and they render their content to the Vizkit3DWidget.

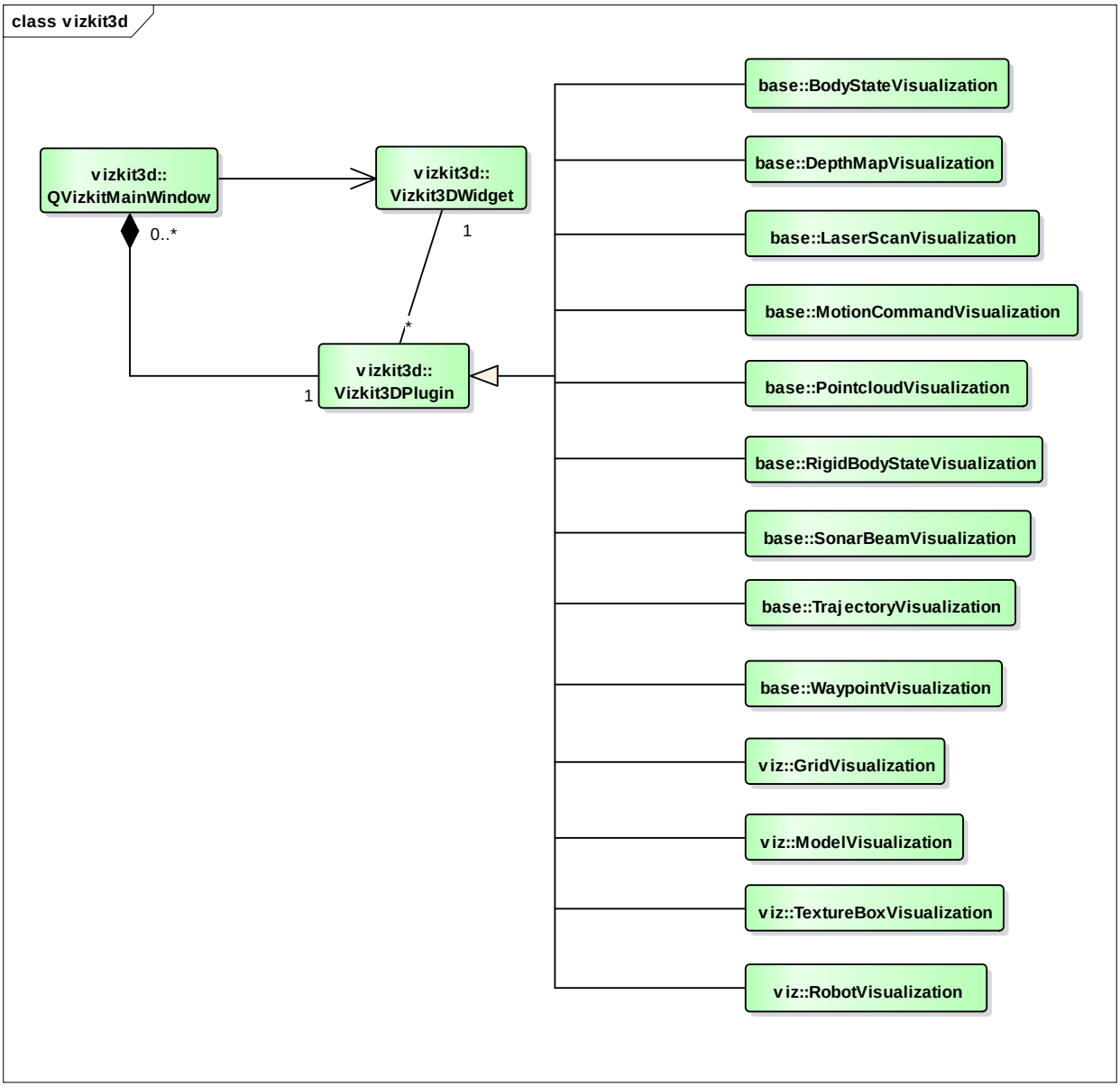


Figure 5-72. Overview of the vizkit3d architecture

The gui/vizkit3d_TASTE library wraps vizkit3d for usage from TASTE models. The architecture of the library is shown in Figure 5-73. It defines the following classes:

- **VizkitInstance:** manages the vizkit3d window initialization, configuration, event loop and termination; the class is a singleton, ensuring that all interactions from the TASTE domain share the vizkit3d window (which must be unique per executable).
- **PluginController and subclasses:** manages a vizkit3d plugin instance; one class is defined for each of the vizkit3d plugins, all inheriting from a base PluginController class that encloses the common functionality.
- **Plugin wrappers:** for each of the base plugins listed above (i.e. for each plugin mapped to a TASTE function) the library defines a C wrapper that exposes the plugin update function to the outside of the library.

- **ConfigurationReader:** supporting class for reading data from a YAML configuration file, inherited by VizkitInstance and PluginController.

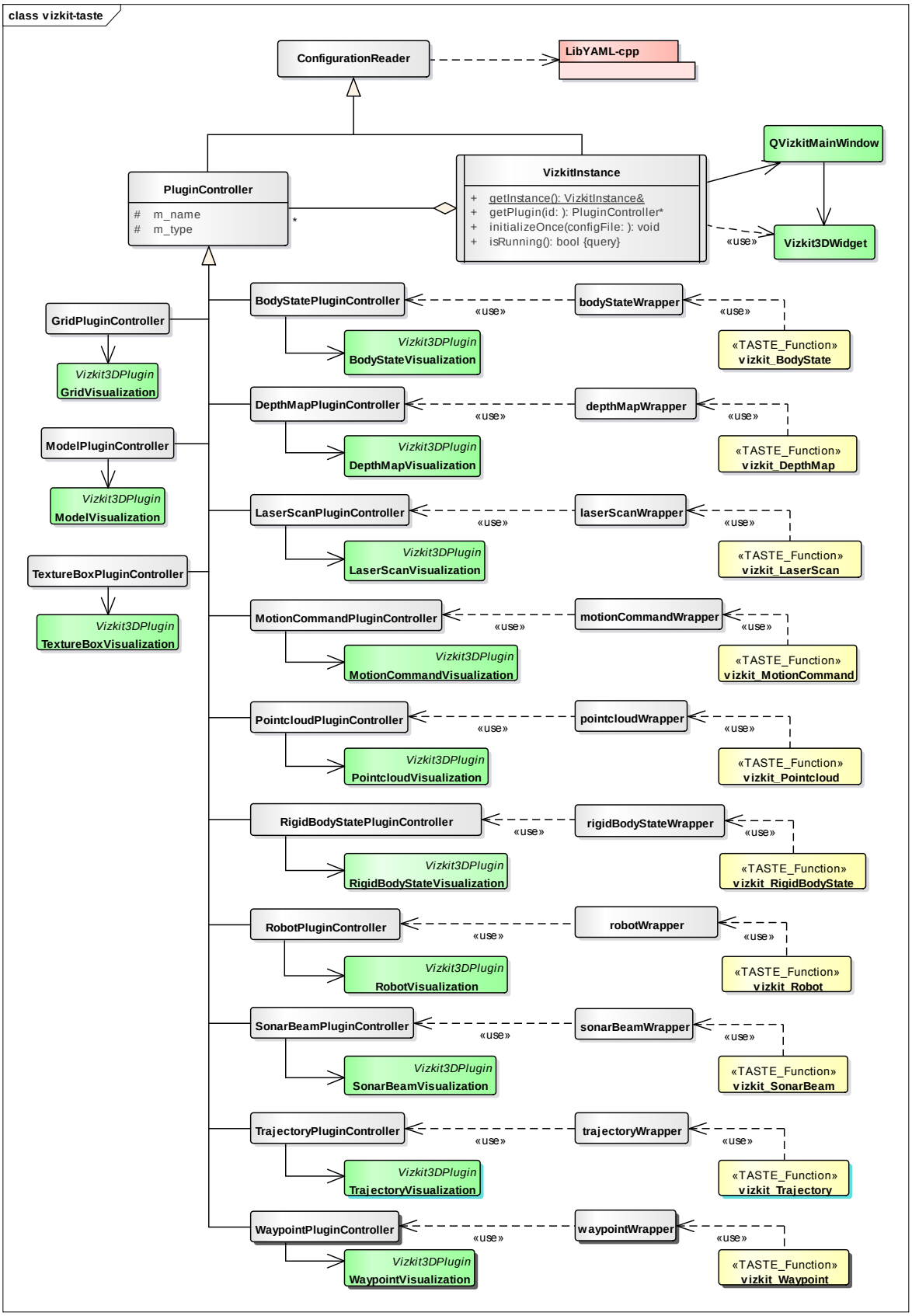


Figure 5-73. vizkit3d_TASTE class diagram

For each of the vizkit3d base plugins, vizkit3d-TASTE contains four elements: the TASTE function, the C wrapper, the controller class and the vizkit3d plugin class itself. The Figure 5-74 details the differences between these four elements for one of the plugins, RigidBodyStateVisualization.

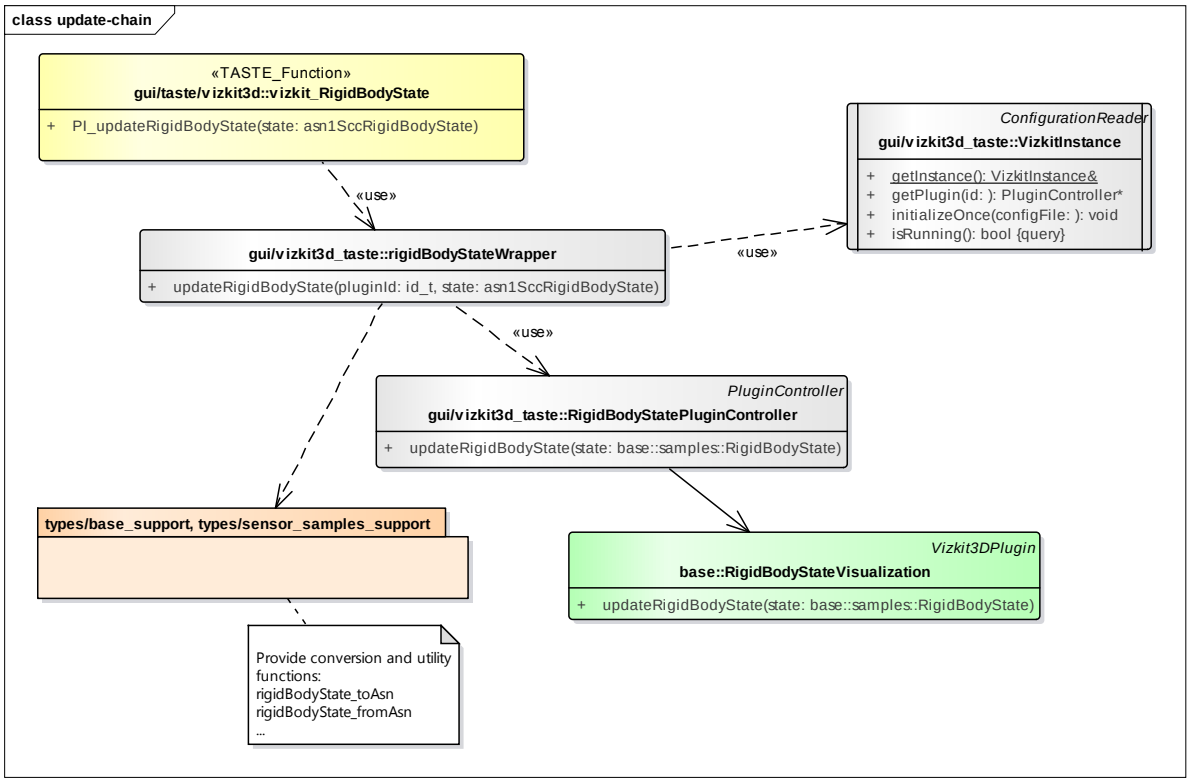


Figure 5-74. Detail of the vizkit3d_TASTE architecture

The TASTE function **vizkit_RigidBodyState** represents one *instance* of the plugin (i.e. if the model contains several objects for which rigid body state data is displayed, it will include several instances of the TASTE function). The plugin instances are differentiated by an identifier that is selected with a TASTE context parameter. The function itself contains one Provided Interface, as per the TASTE Interface View, which gets as input the state as a C data type, **asn1SccRigidBodyState**, generated by the ASN.1 compiler.

The TASTE function uses the C wrapper, **rigidBodyStateWrapper**. The wrapper isolates from the TASTE functions the internal functionality and dependencies of the library. The wrapper provides an update function that dispatches the data to the desired instance of the RigidBodyState plugin according to the identifier, requesting the plugin object from the VizkitInstance. The wrapper also converts the data from the **asn1SccRigidBodyState** C type to the **base::RigidBodyState** C++ type required by vizkit3d, using the functions in the **types/base_support** and **types/sensor_samples_support** packages.

The wrapper uses the **RigidBodyStatePluginController** object, which contains the vizkit3d plugin itself. The controller sets the configuration of the basic vizkit3d plugin according to the YAML configuration file. Finally, the controller class passes the data to the internal vizkit3d plugin object, **base::RigidBodyStateVisualization**, which updates the 3D visualization.

The lifecycle of a TASTE application is managed by the middleware. TASTE functions provide an initialization method that is executed before the function begins reading from its provided interfaces. However, the order in which the initialization methods of the

different functions are run is not guaranteed. This needs to be taken into account for the integration of vizkit3d, which has its own lifecycle tied to the Qt window.

The chosen solution is to implement the vizkit3d instance as a singleton containing the Qt event loop. The instance is initialized by the first plugin function that uses it. The initialization consists on reading the configuration from the YAML file, setting up the vizkit3d window, widget and plugins, and starting the Qt event loop. Successive calls to the initialization function have no effect. This is depicted in Figure 5-75.

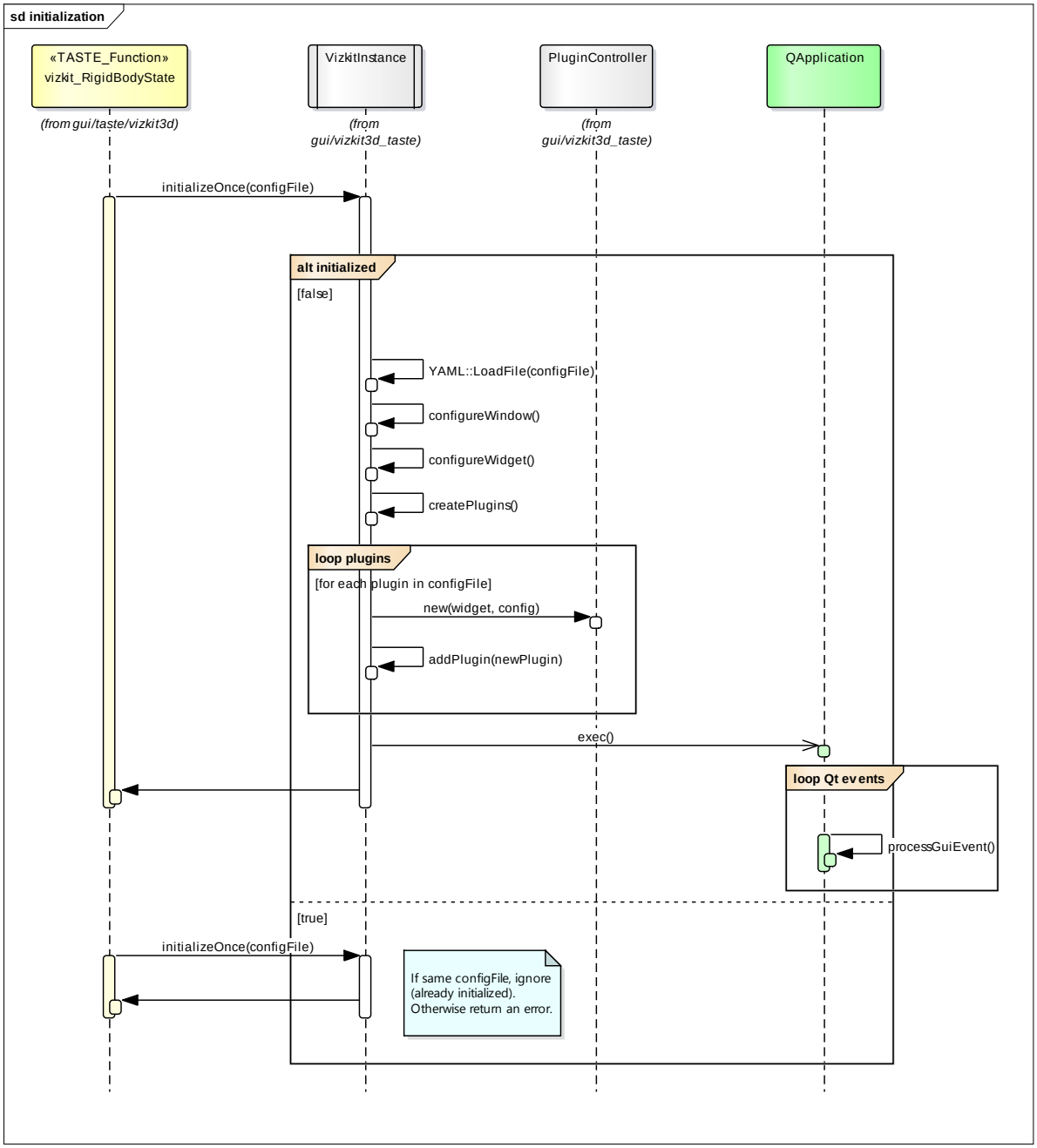


Figure 5-75. vizkit3d_TASTE initialization

The initialization of the vizkit3d plugin is driven by the configuration file and takes place once inside the initializeOnce method. This initializes both the vizkit3d base plugins, which have a corresponding TASTE function, and the visualization plugins that are not visible at the TASTE level.

The TASTE plugin functions will access the internal plugin object based on a plugin identifier defined as a context parameter of the function. The update chain is detailed in Figure 5-76, taking as an example the update of a RigidBodyStateVisualization plugin.

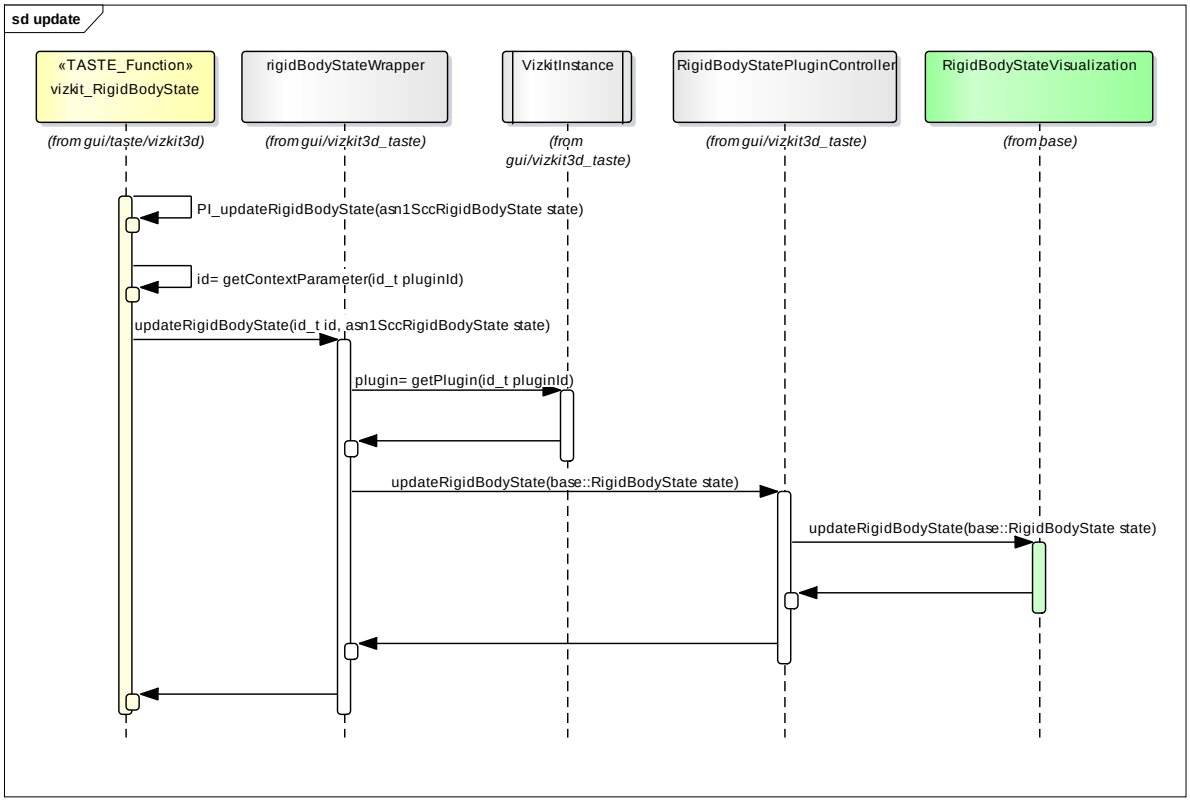


Figure 5-76. vizkit3d_TASTE data update

Upon a call to its provided interface, the TASTE plugin function will read the plugin identifier from its context parameters and invoke the wrapper function. The wrapper requests the plugin object from the vizkit3d instance, performs the type conversion from C to C++, and invokes the update function of the plugin controller. Finally, the plugin controller calls the update function of the internal Vizkit3DPlugin object.

5.5.2.2. INTEGRATION OF ROS ASSETS

Integrating with RViz and Gazebo simulator will require the use of specific data types that are used to visualise data or to exchange data with the simulator.

The minimum set of messages required are the following:

- Geometry messages
- Sensor messages
- Navigation messages
- Visualization messages
- Gazebo messages

These messages will be converted to ASN.1 equivalents using TASTE tools as described in section 5.6.2.1. The transportation of data between TASTE and ROS will be performed by a bridge component as described in 5.6.1.1. That component will map TASTE required interfaces to ROS publishers and provided interfaces to ROS subscribers.

5.5.3.PUS CONSOLE

PUS Console is a tool for debugging, testing and monitoring the sending/reception of telecommand/telemetry packets of the PUS Services standard (see [RD.16]).

The PUS Console is intended for use in the PC representing the Ground Segment, together with other visualization, control and logging tools, during the testing and operations of the robot. The PUS Console allows the operator to interact with the robot using PUS telecommands and telemetry. This mimics part of the operations workflow of a real space mission, and complements the more sophisticated visualization and operation tools used in robotics missions.

The PUS Console provides information of packets sent and received in a table, this table shows important attributes of packets as well as an information field displaying specific information of the packet.

5.5.3.1. SOFTWARE ARCHITECTURE

The application is implemented in Python 3 with some C++ code using *pybind11* library to provide the binding needed between Python and C, since this component uses the C library explained in 5.3.6.1. *Pybind* allows calling C/C++ functions from Python.

Additionally, as the PUS Console should be able to save and load dumps of different tests, the application uses the *sqlite* database library to manage those tasks. The *GUI* is designed using the *Qt* framework, in our case the *PySide* variant, as it provides LGPL-licensed Python bindings for *Qt.* To ease the designing process of *GUIs* we use *QtDesigner,* a graphical tool that allows us to create layouts that can be translated to python code once they are defined.

The application GUI is designed following the MVC (Model-View-Controller) architectural pattern. This pattern divides a given application into three interconnected parts. This allows an efficient code reuse and parallel development.

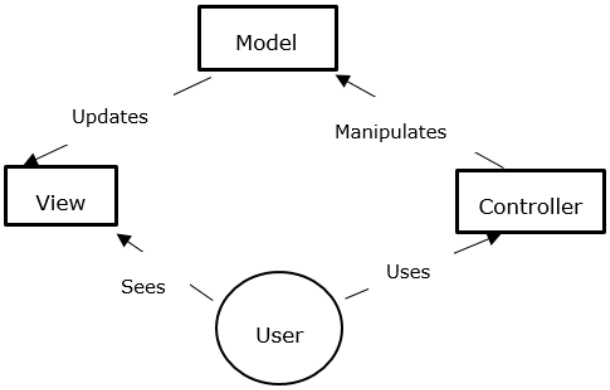


Figure 5-77. MVC diagram

The PUS Console GUI is structured as specified bellow:

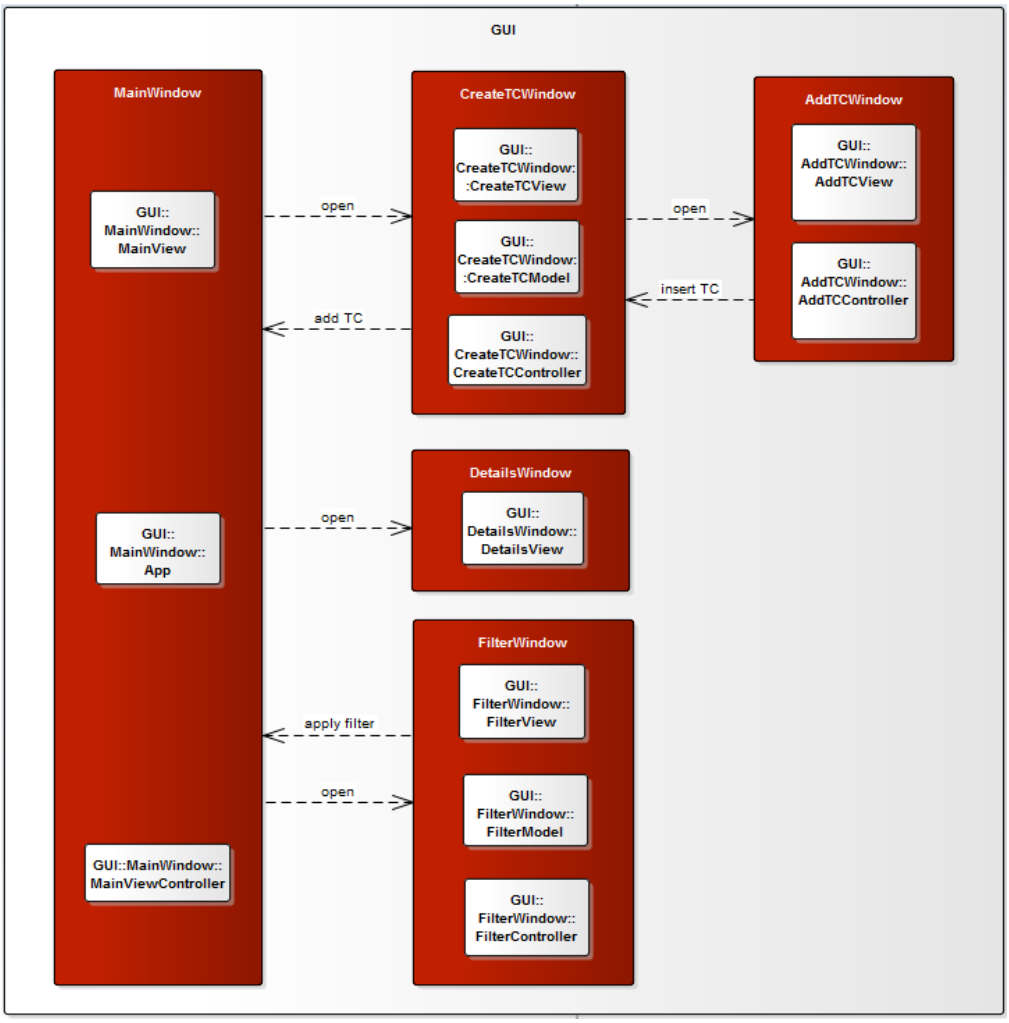


Figure 5-78. Software architecture of the PUS Console GUI

Above figure describes relation between main components of the GUI:

- **MainWindow** is the main window of the application, it shows a table with every telemetry received and every telecommand sent. Additionally it gives options to create and send telecommands, define filters, create dumps of the current state of the table, load dumps and see more detailed information of each packet. As we are using MVC architectural pattern (see Figure 5-77. MVC diagram) this component is split in three classes: *MainView.py* (View), *MainViewController.py* (Controller) and *App.py* (Model).

  The main class of the component is MainViewController, this class is responsible for managing the interaction between the user, the view and the rest of the components.

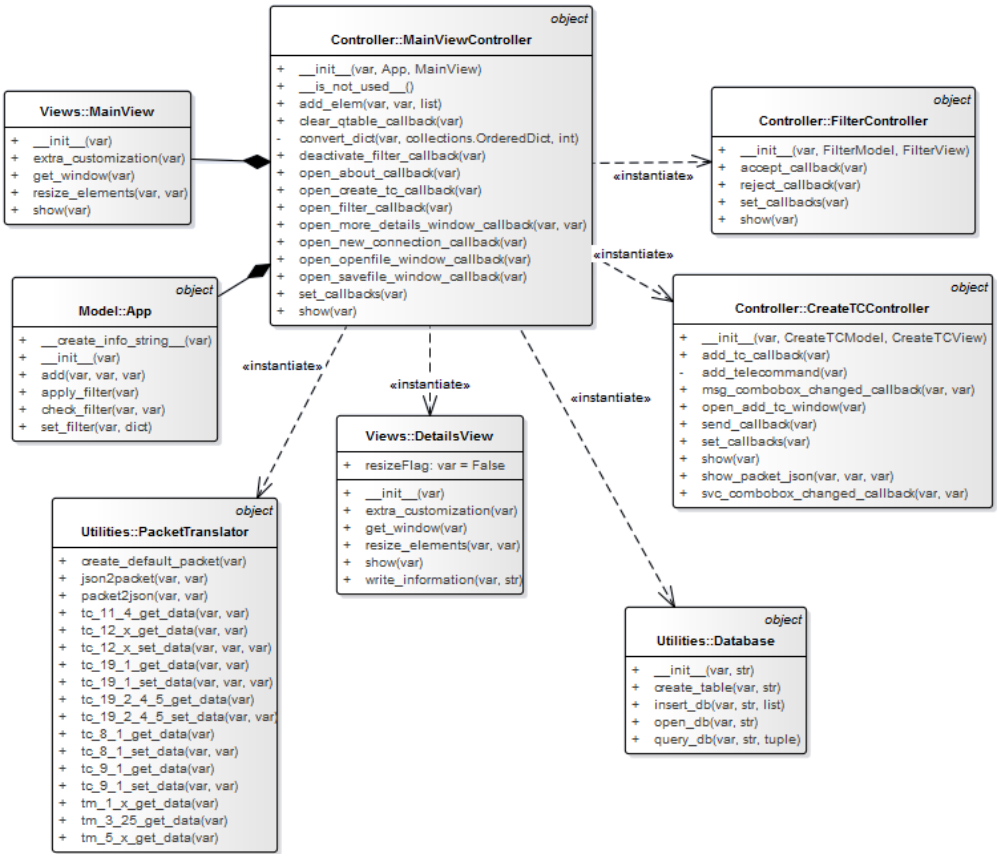


Figure 5-79. MainWindow class diagram

MainViewController instantiates *FilterController*, *CreateTCController* and *DetailsView* when the user clicks the option to open one of those windows. When *DetailsView* is instantiated *MainViewController* also creates a *PacketTranslator* object to parse the selected packet to JSON. Additionally *MainViewController* creates a *Database* object when a dump is to be created or loaded from a *sqlite* database file.

- **CreateTCWindow** is used by the user to create telecommands and send them. This component is divided in: *CreateTCView.py*, *CreateTCController.py* and *CreateTCModel.py*.

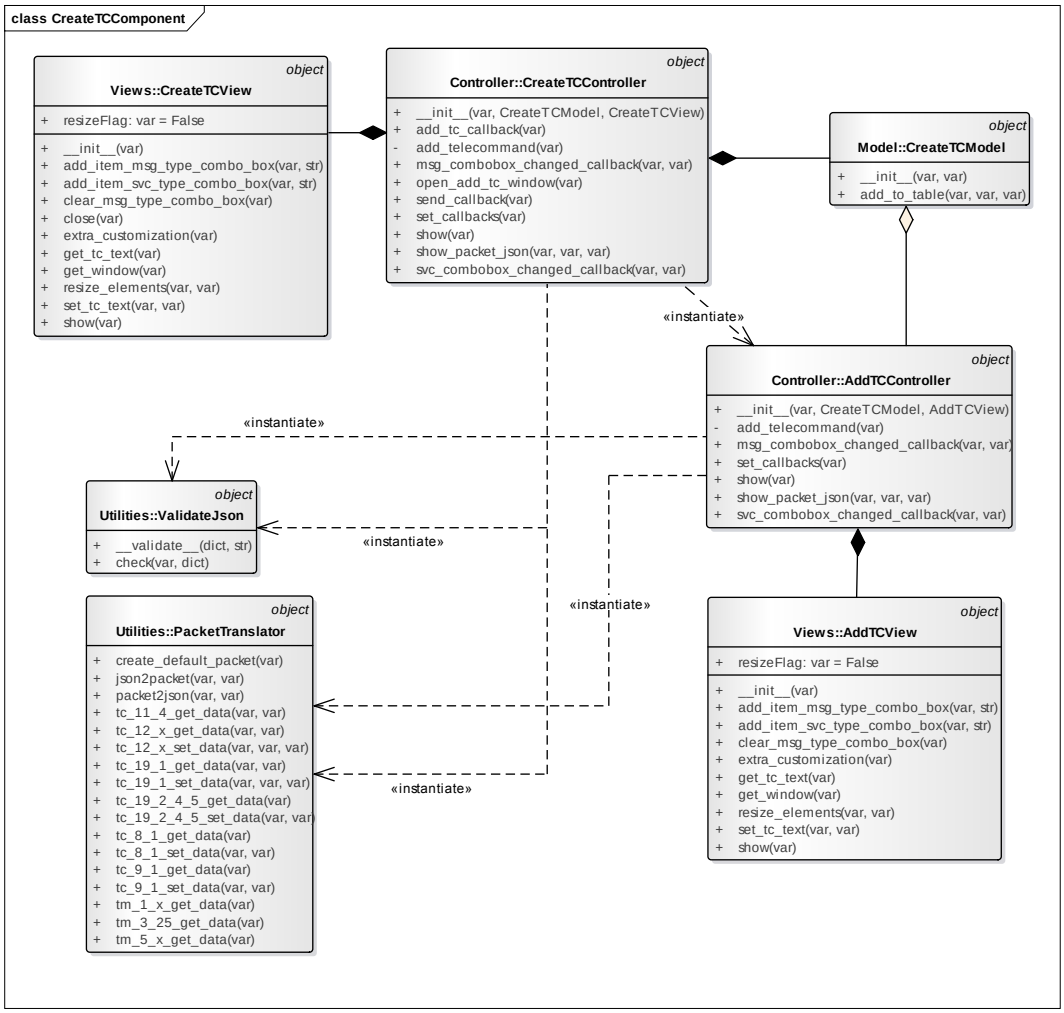


Figure 5-80. CreateTCWindow and AddTCWindow class diagram

Above diagram shows the class structure of CreateTC component and AddTC component. The main class of this component is *CreateTCController*. This class is responsible of the creation of the telecommands that are going to be sent from ground. It instantiates an *AddTCController* object when the user creates a 11_4 or 19_1 packet. This class also instantiates *ValidateJSON* and *PacketTranslator* classes to verify and translate JSON data defined to a correct packet to be sent.

- **AddTCWindow** is used to insert telecommands into certain telecommands. This component is divided in: *AddTCView.py* and *AddTCController.py*.

  This module class diagram is shown in Figure 5-80. CreateTCWindow and AddTCWindow class diagram *AddTCWindow* instantiates *ValidateJSON* and *PacketTranslator* to verify and translate the JSON defined to a correct packet to be sent.

  This module, due to its simplicity and its similarity to *CreateTCWindow* component, share model with that component.

- **DetailsWindow** just shows more detailed information of the packet selected in the table. This component is implemented by *DetailsView.py*.

  This component has neither model nor controller due to the fact it just shows the JSON data received by argument when it is created.

- **FilterWindow** allows the user to define filters to apply to the packet table shown in the main window. Three classes implement this component: *FilterView.py*, *FilterController.py* and *FilterModel.py*.

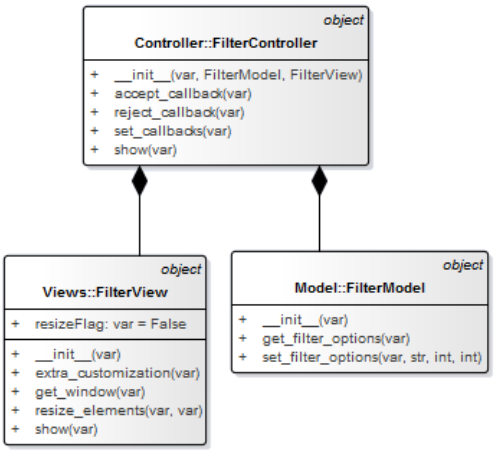


Figure 5-81. FilterWindow class diagram

A general structure of the whole application is shown below.

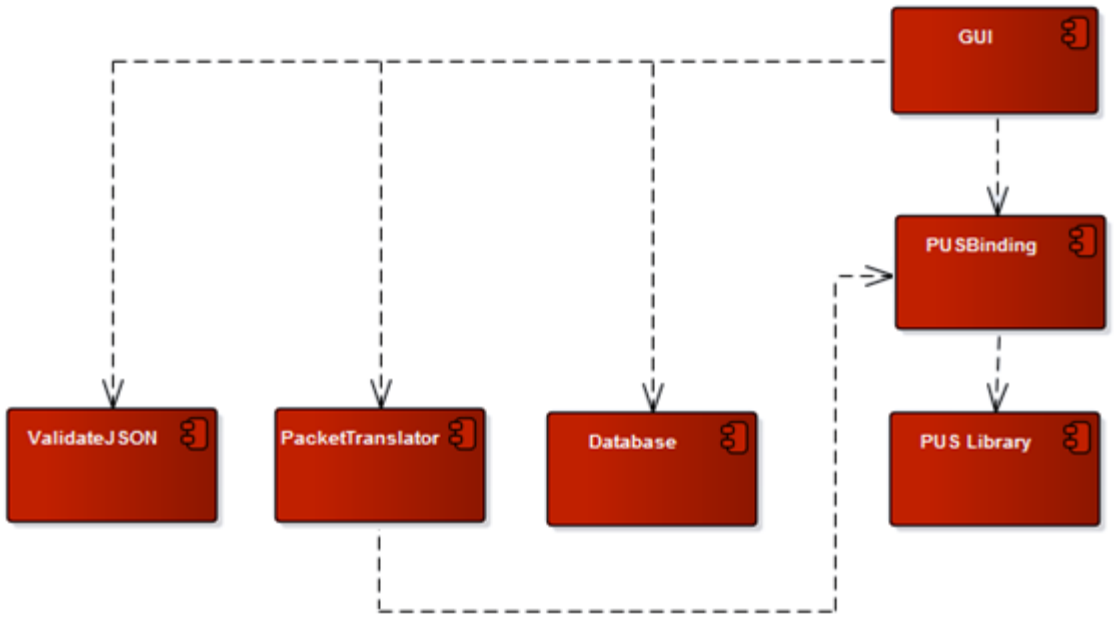


Figure 5-82. PUS Console component diagram

Above figure describes dependencies between the GUI and other modules:

- **PacketTranslator:** This module serves as a parser between the packet representation and JSON. Whenever the user creates a telecommand packet or a telemetry packet is received, the GUI calls functions from PacketTranslator module such as *json2packet* or *packet2json* respectively. This module makes use of PUS Library (see section 5.3.6) functions since it has to be able to create and modify packets with values defined in JSON and vice versa.

- **ValidateJson:** This module uses *jsonschema* to validate each packet JSON representation when creating a certain telecommand. This module requires a folder called *schemas* containing JSON files defining the schema of each telecommand packet.

- **PusBinding:** This C++ module defines bindings with C PUS Library functions using *pybind11* library. Additionally this module defines extra functions to ease the transition between some PUS Library types and Python types and to keep an additional abstraction level with ASN.1 structures.

  C++ code written in *pusbinding.cpp* defines every *struct*, *enum* and function we want to be able to call from python. A small example is shown below.

```
m.def("pus_tc_17_1_createConnectionTestRequest", &pus_tc_17_1_createConnectionTestRequest);
m.def("pus_tm_17_2_createConnectionTestReport", &pus_tm_17_2_createConnectionTestReport);
m.def("pus_st17_createTestResponse", &pus_st17_createTestResponse);
m.def("pus_expectSt17Tc", &pus_expectSt17Tc);
m.def("pus_expectSt17Tm", &pus_expectSt17Tm);
```

Figure 5-83. Code example for Python-C binding

  The sample code binds the C functions defined in the PUS Library for service 17 to Python, so that they can be invoked from a Python script. The first parameter establishes the name that the function will have in Python and the second parameter tells *pybind11* which function to bind to that name by passing the reference of that function.

  A similar mechanism is used to provide bindings for enumerations and structures from C to Python.

  This C++ module that defines the bindings between Python and C has to be compiled and exported as a shared library to be able to access functions from Python.

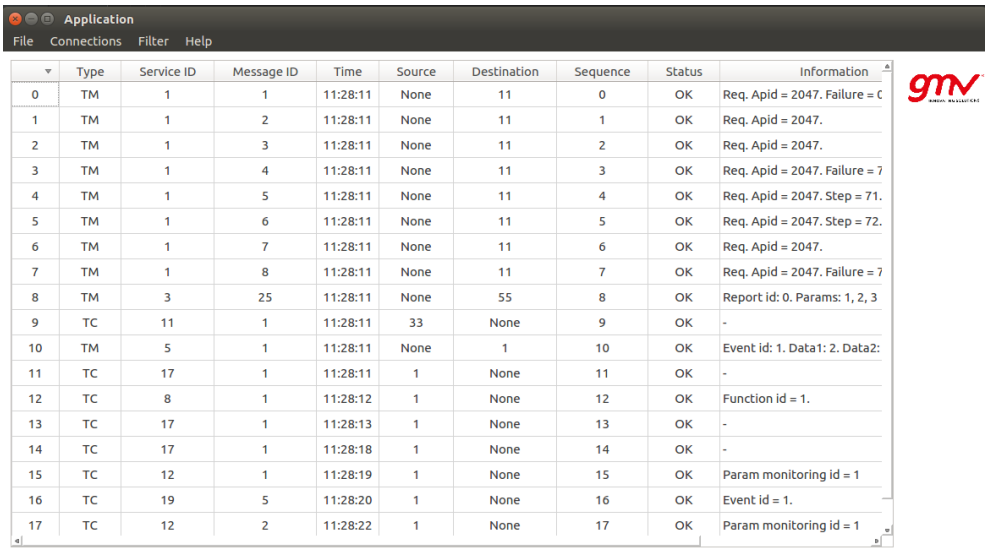
- **Database:** This module implements methods that ease the creation/opening of databases and insertion/querying operations. This module is used by the GUI when a dump of the current state of the table wants to be saved or a dump of a previous state of the same table wants to be loaded. The database manager chosen has been *sqlite*.

5.5.3.2. GUI DESIGN

This section will explain how each window is designed and how windows are connected between them.

Main view

The Figure 5-84 below shows how the MainView screen is designed. The main purpose of this window is showing the user a table with important fields of each telemetry packet received or telecommand packet sent. This table can be sorted by each attribute when the header of a certain column is clicked.



	Type	Service ID	Message ID	Time	Source	Destination	Sequence	Status	Information
0	TM	1	1	11:28:11	None	11	0	OK	Req. Apid = 2047. Failure = C
1	TM	1	2	11:28:11	None	11	1	OK	Req. Apid = 2047.
2	TM	1	3	11:28:11	None	11	2	OK	Req. Apid = 2047.
3	TM	1	4	11:28:11	None	11	3	OK	Req. Apid = 2047. Failure = 7
4	TM	1	5	11:28:11	None	11	4	OK	Req. Apid = 2047. Step = 71.
5	TM	1	6	11:28:11	None	11	5	OK	Req. Apid = 2047. Step = 72.
6	TM	1	7	11:28:11	None	11	6	OK	Req. Apid = 2047.
7	TM	1	8	11:28:11	None	11	7	OK	Req. Apid = 2047. Failure = 7
8	TM	3	25	11:28:11	None	55	8	OK	Report id: 0. Params: 1, 2, 3
9	TC	11	1	11:28:11	33	None	9	OK	-
10	TM	5	1	11:28:11	None	1	10	OK	Event id: 1. Data1: 2. Data2:
11	TC	17	1	11:28:11	1	None	11	OK	-
12	TC	8	1	11:28:12	1	None	12	OK	Function id = 1.
13	TC	17	1	11:28:13	1	None	13	OK	-
14	TC	17	1	11:28:18	1	None	14	OK	-
15	TC	12	1	11:28:19	1	None	15	OK	Param monitoring id = 1
16	TC	19	5	11:28:20	1	None	16	OK	Event id = 1.
17	TC	12	2	11:28:22	1	None	17	OK	Param monitoring id = 1

Figure 5-84. MainView

Additionally, when the user clicks a certain row of the table, **DetailsView** is opened. This window, shown below, displays the whole selected packet in its JSON format.

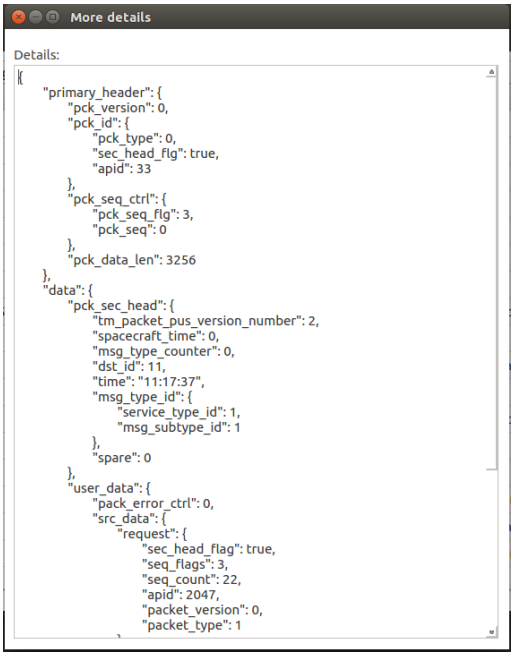


Figure 5-85. DetailsView

MainView has also extra options in the top panel, each of this options provides an specific functionality described below.

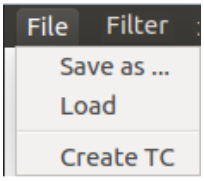


Figure 5-86. File menu

The **File** menu gives the user the following options. When clicking ***Save as …*** option a dialog is opened asking for a name and a path for the dump file to be saved.

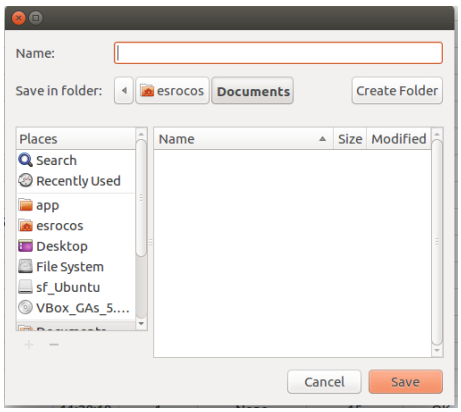


Figure 5-87. Dialog to save a dump file

The screen above retrieves the name and the path given to the dump file and calls **Database** module explained before. This module creates a *sqlite* database in the path specify with the given name.

When clicking ***Load*** option a dialog is opened to select the dump file to be loaded.

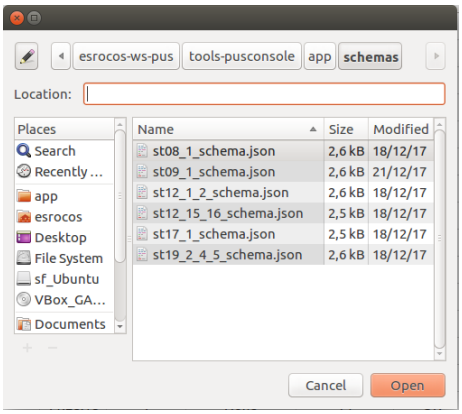


Figure 5-88. Dialog to load a dump file

The screen above retrieves the path to the dump file selected and calls **Database** module that queries *sqlite* database. MainView's table is cleared and the results of those queries are added.

Create TC option opens **CreateTCView** when clicked. This window is explained below.

Top panel menu also gives the user the capability to define filters. When **Filter** menu is clicked the following options are given.

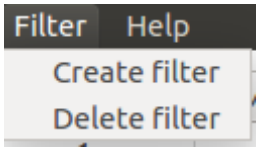


Figure 5-89. Filter menu

Delete filter option removes any filter applied to the table.

Create filter option opens **FilterView** giving the user the capability to define a new filter to be applied.

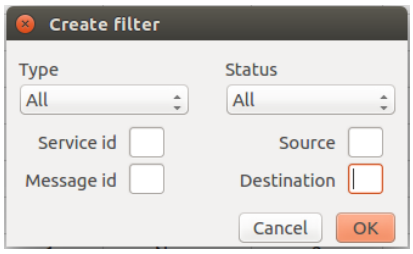


Figure 5-90. FilterView

FilterView allows the user define filters by completing fields shown in figure above.

Type combo box provides the user three options: *All*, *TC* and *TM*. When *All* option is selected every packet, weather it is a telemetry or a telecommand packet, will be shown in the table. If *TC* or *TM* option is selected only telecommand or telemetry packets will be displayed.

Status combo box allows the user to select between erroneous packets, correct packets or every packet.

Additionally, **FilterView** shows four text fields: *Service id*, *Message id*, *Source* and *Destination*. The table will show only packets that satisfy values specified in those fields if a value is given.

Create TC view

This window allows the user to create telecommands and send them. The screen is divided in two sections. The first one, the history, provides the user the capability to recover previous telecommands.

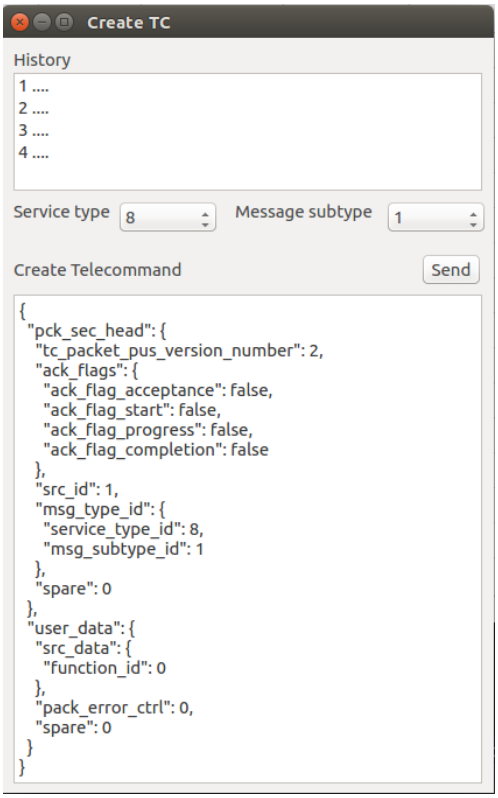


Figure 5-91. CreateTCView

The second one, the creation section, allows the user to create new telecommands. *Service type* combo box contains the service id of every telecommand that the user can send. When a certain id is selected in *service id* combo box, *message id* combo box is filled with the ids available for that specific service.

After the user selects a particular *message id* of a certain *service id*. *CreateTCController* create a default packet of the specified type by calling the related function defined in the PUS Library (see section 5.3.6).

The text box below those combo box shows the JSON of the data section after parsing the created packet using the **PacketTranslator** module. That data section can be modified by the user with the parameters needed.

Once the user clicks send, the **ValidateJson** module checks if every field of the JSON defined is correct. If that happens, the **PacketTranslator** module creates a packet object from the JSON and the telecommand is sent and added to the table, otherwise a warning message will be displayed telling the user that the JSON specified is incorrect.

Add TC view

PUS services 11_4 and 19_1 allow embedded telecommands. This screen let the user to define those particular telecommands.

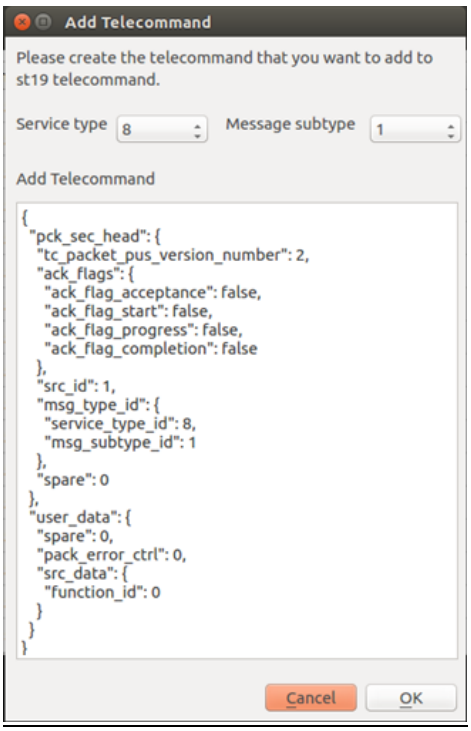


Figure 5-92. AddTCView

The process of showing data section of a certain packet is similar to the process described for the Create TC view. Once the user clicks OK, weather the JSON has been changed or not, **ValidateJSON** is called to verify the packet correctness, in the case it is correct, **PacketTranslator** creates a packet from that JSON and it is returned to **CreateTCWindow** that inserts that packet inside 11_4 or 19_1 packet it is being defined.

Additional views

In addition, a window showing certain information of specific services such as *housekeeping* or *OBCP* will be added. The design of this window is not fixed yet.

5.6. INTEGRATION OF LEGACY SOFTWARE

ESROCOS provides support for importing and exporting software components from the ROS and ROCK frameworks, in order to facilitate the migration of legacy code to the framework.

The integration with ROS and ROCK is done at two levels. At runtime, a mechanism of middleware bridges is provided to allow for the communication of software components running in the PolyORB-HI middleware of TASTE with those running on an external framework. At development time, tools to import ROCK and ROS data types to the ASN.1 language used in TASTE are implemented. The imported types are used in the middleware bridges and can be also used by the developer to integrate existing software from the ROS and ROCK ecosystem at library level.

Finally, a tool to export TASTE models to the ROCK framework is provided. This tool is intended to give a first contact with TASTE to ROCK users. A similar tool for ROS is not implemented, as there is no such clear correspondence between TASTE and ROS modelling concepts.

5.6.1. MIDDLEWARE BRIDGES

5.6.1.1. OVERVIEW AND TASTE INTEGRATION

In order to allow for interoperation of systems built with ESROCOS with existing robotics application components and tools, in particular from the ROCK and ROS ecosystems, ESROCOS provides the capability of interconnecting the PolyORB-HI middleware with the domains of the middleware used in those frameworks, using a bridge component.

The purpose of the bridge is to map the message abstractions of the corresponding middleware and forward the communications between the PolyORB-HI and the external domain. The figures below illustrate the principle.

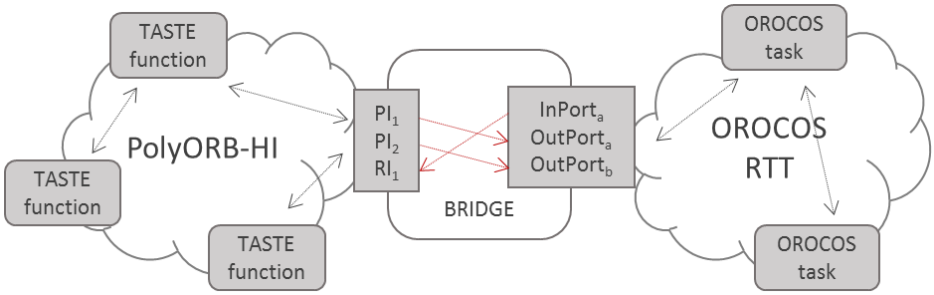


Figure 5-93. TASTE-ROCK bridge

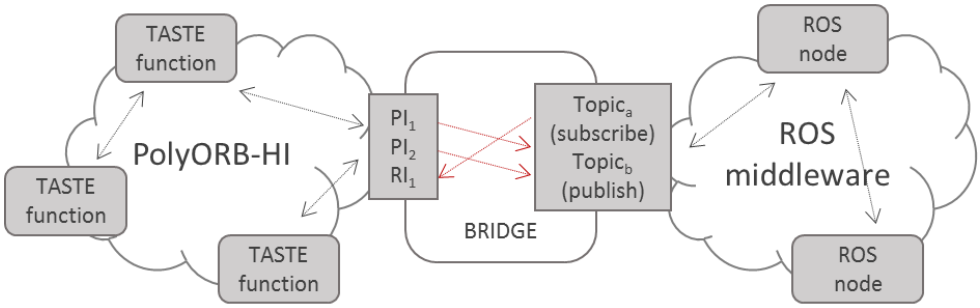


Figure 5-94. TASTE-ROS bridge

The bridge component is visible in the two middleware environments, and provides a set of desired interfaces that are translated and forwarded from one environment to the other.

The advantage of the bridge approach is that the existing components can run unmodified. This should ease the maintenance of the software and the integration with existing developments and tools.

A bridge component is modelled as a regular TASTE function in the Interface View. In order to enable the creation of the bridge code, the corresponding function must be selected by means of its Context Parameters. A set of ASN.1 types, grouped in **ESROCOS-Directives**, are used for this purpose. The definitions of these types are shown below.

```
-- H2020 ESROCOS Project
-- Company: GMV Aerospace & Defence S.A.U.
-- Licence: GPLv2

ESROCOS-Directives DEFINITIONS ::=

-- ESROCOS directives are used in the Interface View to define Context
-- Parameters of functions. These parameters are read by ESROCOS tools
-- and used to trigger certain functionalities or code generation steps.

BEGIN

    IMPORTS T-Boolean  FROM TASTE-BasicTypes;

    -- Enable generation of a TASTE-ROS bridge component from the function
    ESROCOS-ROS-Bridge ::= T-Boolean;

    -- Enable generation of a TASTE-ROCK bridge component from the function
    ESROCOS-ROCK-Bridge ::= T-Boolean;

END
```

A function in the Interface View can define a Context Parameter of any of these types, indicating whether a middleware bridge should be generated from it. This is similar to the TASTE Directive mechanism used to define function-specific compiler flags in TASTE. By default, if no ESROCOS directive is present no bridge will be generated.

Not all TASTE functions or interfaces, however, can be mapped to an external middleware. The following constraints must be met:

- Only sporadic interfaces, either provided or required, can be mapped; bridge functions cannot contain cyclic, protected or unprotected interfaces (this is similar to GUI functions).
- The implementation language for the bridge functions is constrained and depends on the desired type of bridge:
  - ROS bridge components must be GUI functions, as the ROS bridge relies on the TASTE-Python interface code generated for GUIs.
  - ROCK bridge component must be C++ functions, because the generated bridge code is C++.

These constraints are checked when the bridge code generation is requested.

The generation of the middleware bridge code is implemented in Python. The function definitions and the applicable directives are read from the **interfaceview.pro** file of the model. Figure 5-95 outlines the architecture of this software.

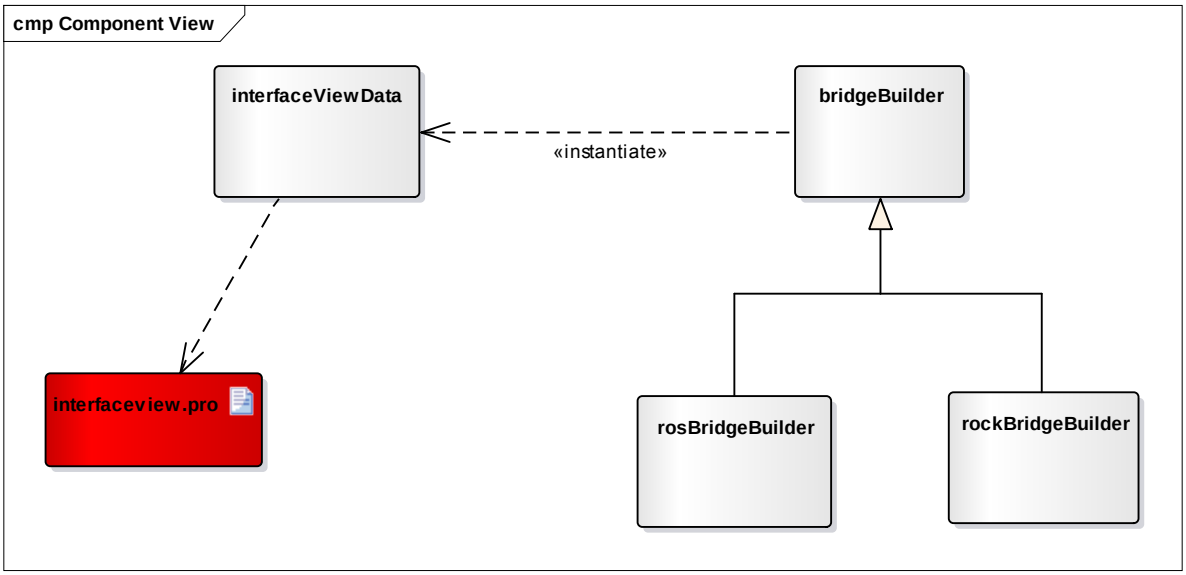


Figure 5-95. Middleware bridge creation architecture

A builder class is defined for each type of middleware bridge, namely ROS and ROCK. The bridgeBuilder class contains the common functionality. This class instantiates the interfaceViewData object, which parses the interfaceview.pro file to read the model information. The builder classes use this information to generate the code (Python or C++) that implements the bridge.

5.6.1.2. TASTE-ROS BRIDGE

In order to connect the TASTE developments with ROS a bridge component will be required. This bridge component will be responsible to transport data from TASTE to ROS and vice versa. Data will be modelled in ROS messages and their equivalent ASN.1 data types as described in sections 5.5.3 and 5.6.2.1.

The bridge will be automatically generated based on input from TASTE interface view. The user will create a component with a special bridge type in a similar manner as it is done for the GUI component. Then TASTE will automatically generate the required code for publishers and subscribers, along with any translation required at the message level using the *data management tools* and *build support* that are called from the orchestrator during the project compilation.

The whole automatic code generation process will follow the paradigm for creating a GUI component, where data is pushed in buffers and visualised using an extra application.

5.6.1.3. TASTE-ROCK BRIDGE

The interface between TASTE and ROCK components is provided by a C++ class generated specifically for each TASTE function with the ESROCOS-ROCK-Bridge directive set to true. For a TASTE function named **FunA**, a class **FunA_RockBridge<T>** is created, where **T** is an adaptor class, defined by the user, which provides conversions between ASN.1 and C++ types.

Figure 5-96 below shows the architecture of the bridge code generated for the sample function **FunA**, with two sporadic interfaces, one provided (*funA_PI_pi1*), and one required (*funA_RI_ri1*). The figure shows in red the code to be provided by the user, in

white the code generated by TASTE and the bridge creation, and in grey the RTT library classes.

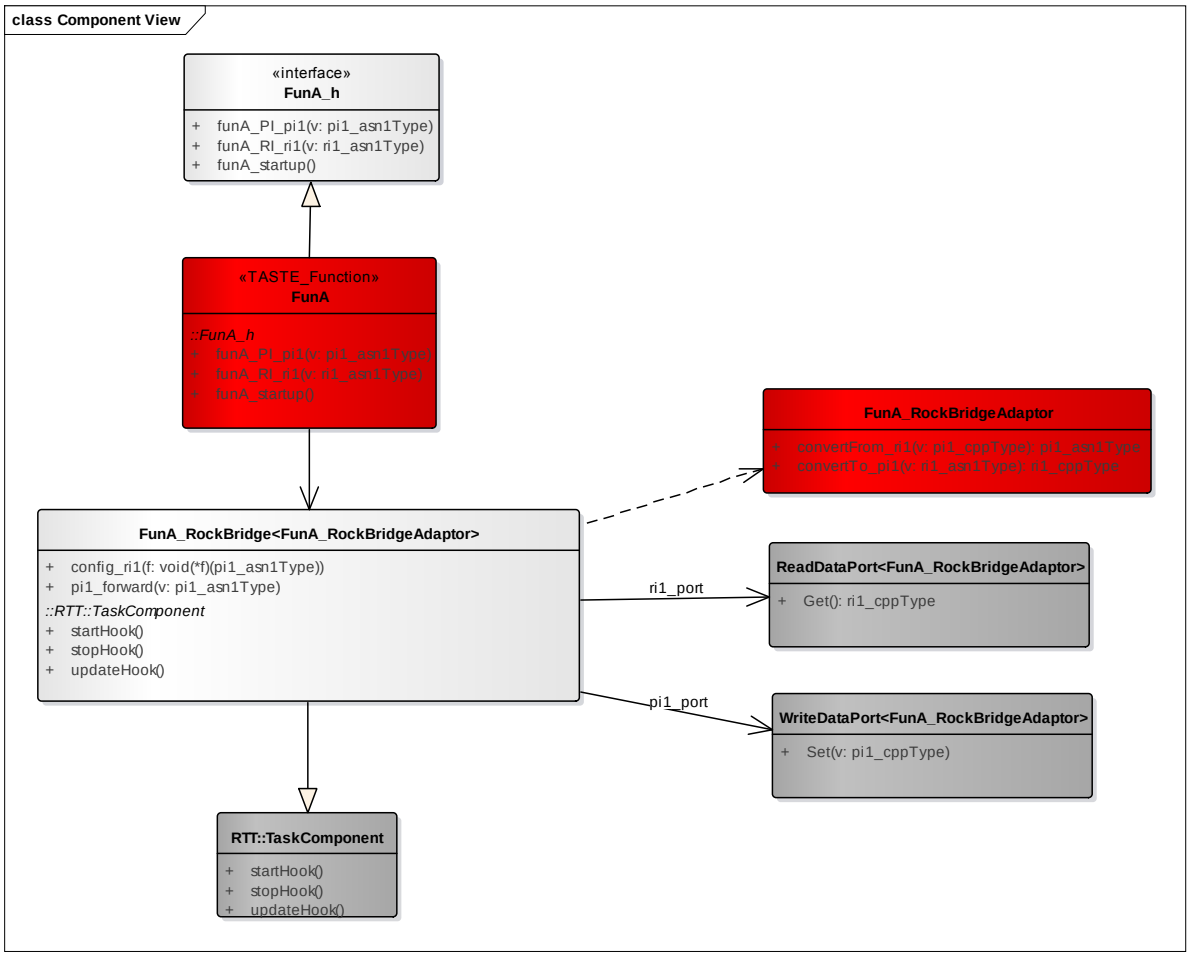


Figure 5-96. ROCK bridge architecture

The TASTE code skeleton **FunA_h** is generated by TASTE, and the user must provide the implementation of **FunA** which uses the bridge class to interface with ROCK.

The bridge generation outputs the class template definition for **FunA_RockBridge<T>**. This template must be instantiated with an adaptor class provided by the user, **FunA_RockBridgeAdaptor**. This class must provide the methods to convert between the ASN.1 and the C++ representation of the types. ESROCOS provides a set of conversion functions already implemented for many common data types (see section 5.3.1), so in most cases the adaptor class will just invoke these functions.

At start-up, **FunA** must instantiate the bridge class and configure it, supplying a pointer to the *funA_RI_ri1* function that will be invoked upon receiving any data through the corresponding ROCK port. This must be done for all the required interfaces of the function.

The provided interface *funA_PI_pi1* uses a direct function call to forward the request from TASTE to the ROCK environment. TASTE automatically generates a method for each provided interface, and the user must edit the implementation of this method and invoke *pi1_forward* to send the message to ROCK. This must be done for all the provided interfaces of the function.

The **FunA_RockBridge** class implements the **RTT::TaskContext** interface, which manages the connection to the ROCK environment and implements the task lifecycle expected by ROCK. It defines two data ports, an **RTT::ReadDataPort** for *ri1*, and an **RTT::WriteDataPort** for *pi1*. The activation of the function is port-driven, i.e., the

updateHook method is executed whenever data is available at the *ri1_port*. The forwarding of messages between TASTE and ROCK is only allowed when the task is in "running" state, which is managed by the *startHook* and *stopHook* methods.

When a message from TASTE is received through the *pi1* provided interface, the user-provided code calls *pi1_forward*. This function converts the data type using the method *convertTo_pi1* of **FunA_RockBridgeAdaptor**. Then, it sends the message out through RTT using the *Set* method of *pi1_port*.

When a message from ROCK is received through the *ri1* port, the **TaskContext** triggers the execution of the *updateHook* method. This method retrieves the message from *ri1_port* using its *Get* method, converts it to ASN.1 using the *convertFrom_ri1* method of **FunA_RockBridgeAdaptor**, and calls the *funA_RI_ri1* function using the function pointer configured by the user code.

5.6.2. FRAMEWORK IMPORT TOOLS

In order to enable the creation of middleware bridges for the ROS and ROCK frameworks, ESROCOS includes some capabilities to import the data types defined in those frameworks, transforming them in equivalent ASN.1 data types usable by TASTE, and generating conversion functions to allow transforming the data between the TASTE and the external representation.

In addition to being used in the middleware bridges, the type import capabilities can also be used to integrate existing libraries or software components at the source code level. The type import tools are made available to the user for this purpose.

5.6.2.1. MAPPING OF ROS TYPES TO ASN.1

As already mentioned in section 5.5.3, a set of messages is required for TASTE to interact with ROS. These messages will be translated to ASN.1 equivalent data types in an automated way during the project creation.

A tool that automatically translates a list of user defined ROS messages to ASN.1 data types has already been created and integrated to TASTE. The tool is capable of translating basic and composite data types by handling message dependencies automatically.

The tool is invoked by passing the correct flags to the TASTE project creation scripts.

5.6.2.2. MAPPING OF ROCK TYPES TO ASN.1

In order to translate ROCK data types to ASN.1 equivalents, ESROCOS relies on OROCOS' **typelib** library. This type introspection library is normally used to parse C++ type definitions and produce IDL (and other) mappings for transference over the RTT middleware.

The SARGON project implemented an import tool to create TASTE models from ROCK, including importing the ROCK C++ types to ASN.1. This tool used a C++ parser built on Java.

Instead of reusing and extending this tool, it has been preferred in the frame or ESROCOS to use an alternative approach and rely on the type parsing mechanisms already used inside OROCOS. By using the existing type introspection library of OROCOS, it is ensured that all type structures supported by OROCOS can be mapped to ASN.1 (with the restriction that data types in TASTE must have a fixed size).

The SARGON project also implemented an Orogen file parser to transform the OROCOS component definitions into TASTE models. This transformation is incomplete, as Orogen files are actually Ruby scripts that can have arbitrary functionality that cannot be mapped to TASTE models. The transformation of Orogen files to TASTE has been de-scoped in ESROCOS.

typelib is a C++ library that provides Ruby bindings. Two alternative approaches are considered to implement the transform between C++ and ASN.1 types using this library:

- Extend **typelib** to support ASN.1 as an output format

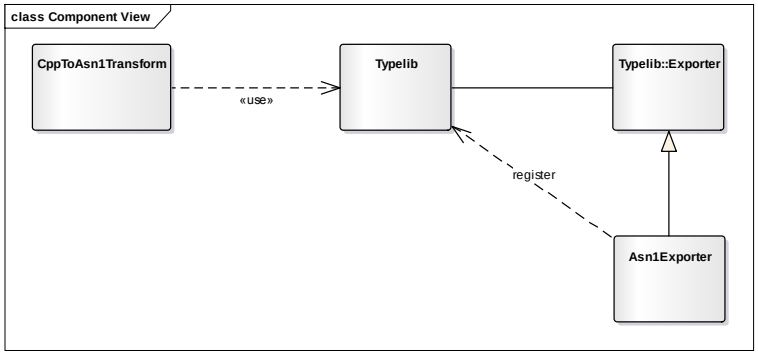


Figure 5-97. ROCK types import using Typelib export plugin

- Use **typelib** to transform the C++ types into typelib's XML representation, and then process this XML to produce the ASN.1 output

Figure 5-98. ROCK types import using Typelib XML

These two approaches will be traded-off at the beginning of the implementation phase. Additionally, the possibility to generate type conversion functions between the C++ and ASN.1 (C) types using the same mechanism will be assessed.

5.6.3.FRAMEWORK EXPORT TOOLS

ESROCOS includes the TASTE2rock tool, originally developed in the SARGON project and updated to be compatible with the version of TASTE used in the project. The purpose of the tool is to provide a path for ROCK users to start using TASTE and taking advantage from its analysis capabilities while remaining compatible with their legacy software.

Nevertheless, it must be highlighted that the output of the TASTE2rock transform uses the RTT runtime in contrast to ESROCOS, which runs on PolyORB-HI. As a consequence, the TASTE2rock transform does not provide code for RTEMS platforms or direct add-ons to the existing safety critical environments for the space qualified environment.

The table below presents the mapping to ROCK of each of the elements in the TASTE Interface View:

Table 5-7. Mapping of TASTE to ROCK concepts in TASTE2rock

TASTE IV element	ROCK transform	Remarks and limitations
Component	None	Components are helping elements that do not translate to runtime artefacts. In TASTE 2.0, components have been replaced by nested functions.

TASTE IV element	ROCK transform	Remarks and limitations
Function	Task context	Each function is characterized by its interfaces, properties and internal state. Each function is mapped into a ROCK task context. The activation mode of the ROCK task is port-driven. Cyclic PIs are handled by adding extra elements. This allows to combine sporadic and cyclic behaviour in the task context, which is not normally possible in ROCK.
Sporadic PI	Input port + data type	ROCK input ports receive one parameter. For each asynchronous PI, the transform generates a struct type with all the PI parameters (if the PI has zero parameters, a dummy parameter is inserted, for the shake of compatibility with ROCK). The generated types must be shared by all the functions that provide and require the interface, so they are placed in a shared type library that can be imported by any component. One limitation of this approach is that the types are generated per interface, so two interfaces will result in two different types even if they have the same signature.
Cyclic PI	Input port + data type + activator task	Cyclic PIs are mapped in the same way as a sporadic PI with zero parameters. In addition, an activator task is generated. This is a task context with periodic activation (with period defined in TASTE) and one output port. When a deployment is generated with TASTE2rock, this output port is connected to the input port of the function so that it is called periodically.
Sporadic RI	Output port + data type	Each sporadic RI is mapped to an output port with the same data type as the matching PI.
Protected PI	Operation + data types	Protected PIs are converted into task operations, set to execute the method by the providing task (hence providing mutual exclusion between callers). In addition to the operation, ROCK data types are defined for the input and output parameters.
Unprotected PI	Operation run by caller thread + data types	Unprotected PIs are converted into task operations, set to execute the method by the caller (hence not providing mutual exclusion). In addition to the operation, ROCK data types are defined for the input and output parameters.

TASTE IV element	ROCK transform	Remarks and limitations
Context parameter	Property + data type	Context parameters in TASTE are typed values that are defined in the IV and accessed at runtime through a context constant. If a function has context parameters, a configuration type is generated with one field per parameter. A ROCK property "config" of this type is added to the task context. In addition, the needs configuration flag of the ROCK task is set. The parameters are grouped in a single type accessible in the same library as the shared interface types. One difference between TASTE context parameters and ROCK task properties is that the former are set at the architecture modelling stage, while the later can be set at initialization or runtime. The initial value of the "config" property reflects the values set in the TASTE Interface View for the function. However, when TASTE-generated code is linked to the ROCK task, the internal context constant is used. This means that changes done to the ROCK "config" property at initialization or runtime cannot be reflected to the context and are not visible to the code.

The TASTE to Rock transform uses a procedural approach to generate, using templates, the software structure and source code of a set of Orocos entities (types, tasks, etc.) that correspond to a model described by a TASTE Interface View.

The transform application is to be implemented in Python. It is invoked using a TASTE2Rock.py script. It receives as input the products of the C code generation by TASTE (including the iv.py file with the Interface View in a format easily accessible from Python code), and exports one of the following:

- A Rock library component containing the types used by the interface and configuration properties of the system.
- A runnable Rock component for each of the functions in the Interface View.
- A Rock deployment and start script for all the functions in the Interface View.

The generated component skeleton can be built and installed with the usual Rock tool, amake. The build requires the C header files and binaries generated by the TASTE C build.

Figure 5-99 presents an overview of the architecture of the TASTE2Rock application.

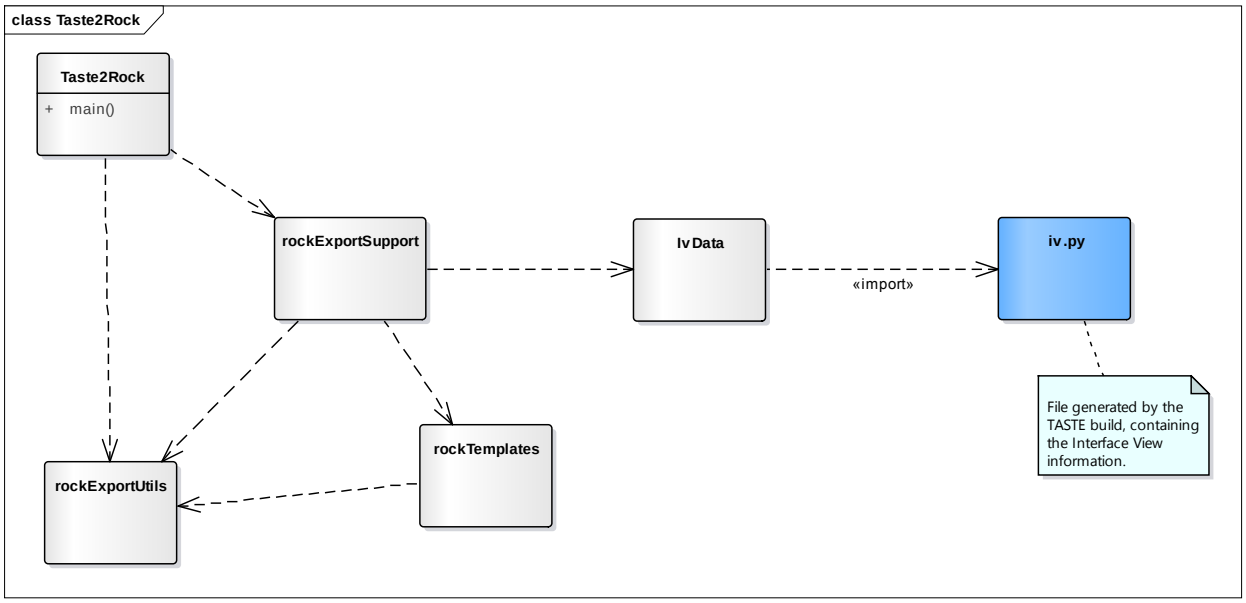


Figure 5-99. TASTE2Rock general architecture

Where the depicted components are:

- **TASTE2Rock:** main program.
- **rockExportSupport:** module that implements the overall logic of the transform for the different TASTE architecture elements.
- **rockTemplates:** module that loads the source code templates needed by the transform.
- **ivData:** class IvData to extract the model information from the TASTE-generated **iv.py** file, which contains the information of the Interface View.
- **rockExportUtils:** additional support functions not related to the transform functionality (error handling, etc.).

The TASTE2Rock application uses a set of templates to generate the source code and some other files of the output Rock system. To implement these templates, the Mako library has been chosen. Mako is a Python library for generating text files from template files that contain labels, logic (conditional, loops, etc.) and Python code.

The rockTemplates module loads a set of Mako template files, as shown in Figure 5-100. These templates receive an instance of the IvData class, so that they can traverse the data in the Interface View model.

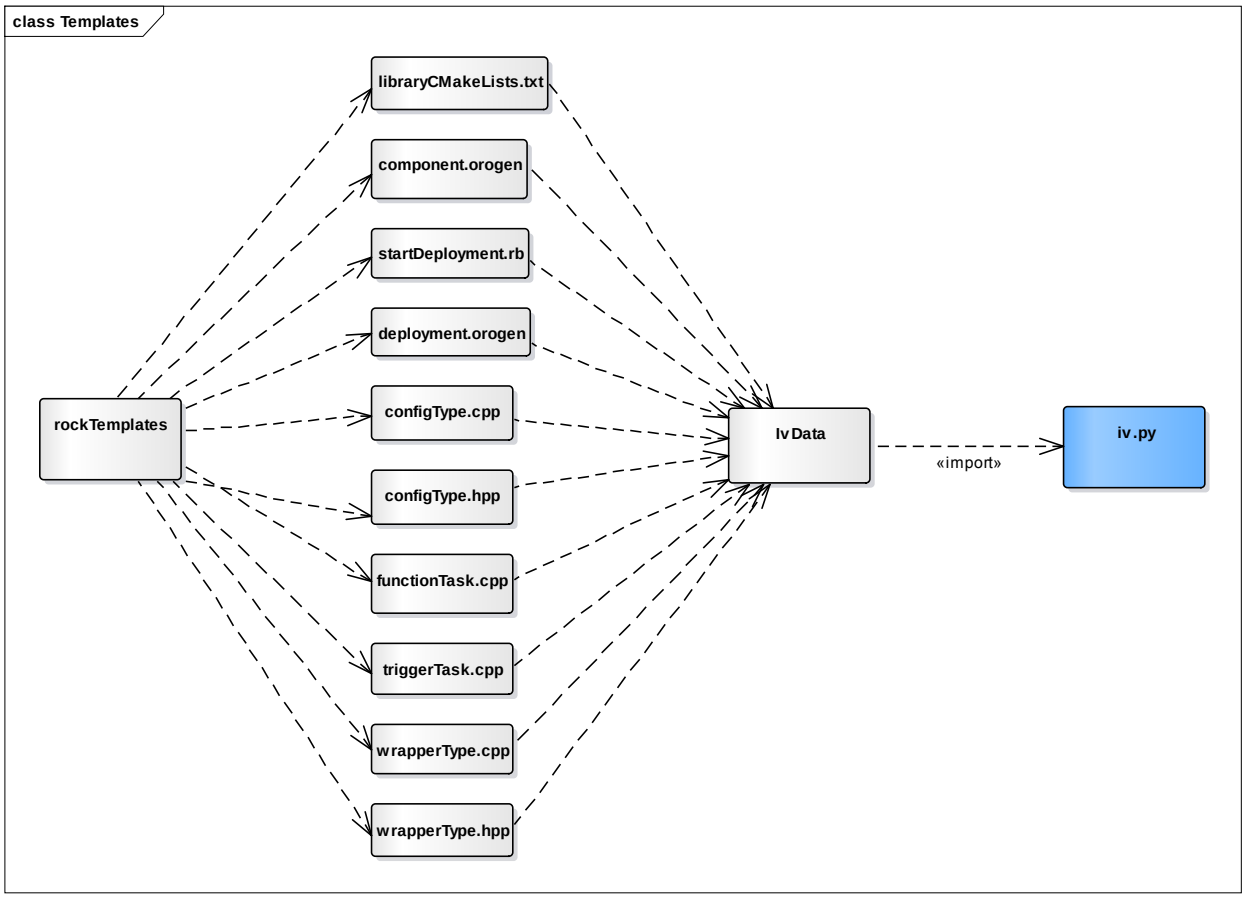


Figure 5-100. TASTE2Rock template architecture

The transform process uses the following file templates:

- For the creation of a type library:
  - *wrapperType.hpp/.cpp*: C++ class wrapping an ASN.1 defined type used in a TASTE provided/required interface
  - *configType.hpp/.cpp*: C++ class wrapping the ASN.1 defined context parameters (properties) of a TASTE function
  - *libraryCMakeLists.txt*: build file for the type library, defining the include paths for the TASTE-related dependencies
  - In addition, the manifest.xml file of the Rock component is created programmatically.
- For the creation of a component mapping a TASTE function:
  - *component.orogen*: definition file for the component
  - *functionTask.cpp*: body of the C++ class, inheriting from RTT::Task, that contains the behavior of the function
  - *triggerTask.cpp*: body of the periodic activator task for a TASTE cyclic provided interface

    In addition, the CMakeLists.txt file is programmatically patched to add the TASTE-related include paths and library dependencies.
- And for the creation of a Rock deployment for the transformed system:
  - *deployment.orogen*: definition file for the deployment

- *startDeployment.rb*: Ruby launch script for the deployment

The TASTE2Rock application relies on the Rock infrastructure to perform its work. In particular:

- It uses the Rock **component templates**, which are provided in a local Git repository under the Rock install;
- It runs the **orogen** tool to generate the skeleton of the output Rock components;
- And it uses the Autoproj build utility **amake** to build and install the generated components.

As the TASTE2Rock export process relies on Orogen and amake, the exported components are regular Rock components installed locally. This means that they provide change tracking with Git and dependency management with Autoproj, and that they can be referenced and managed from Orocos Ruby scripts.

The Figure 5-101 presents an overview of the transform process. The TASTE2Rock application relies on the header and object files generated by the TASTE build process. It then uses the Rock templates and utilities to generate a set of Orocos components and configure them under the local Rock install.

The steps of the transform are the following:

- Prepare the directory structure of the desired components, using the Orocos directory templates from the local Rock install.
- Generate the component source skeletons using Orogen and patching the makefile.xml and CMakeLists.txt files as appropriate.
- Populate the code of the components and types from the Interface View data exported by TASTE (iv.py). The Deployment View information, which must be manually exported from TASTE, is used to determine the name of certain TASTE-generated functions.
- Build the component code, together with the implementation of the TASTE functions.

In order to make available to the Orocos components the functions created with TASTE, it is necessary to build a library containing these functions. This can be done manually using the "ar" tool, but a small application CreateTASTELib is provided for convenience. It might be also necessary to link additional user-developed code.

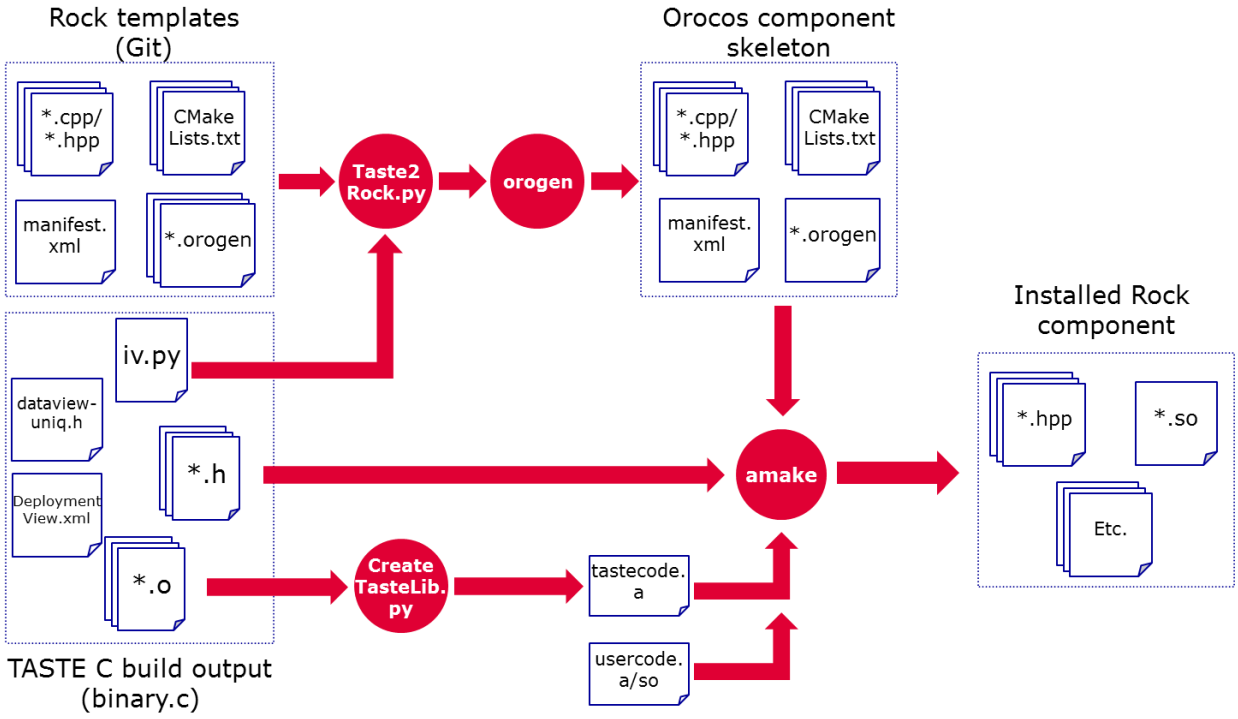


Figure 5-101. TASTE2Rock transform logic

The mapping of the TASTE abstractions to Rock elements is done as specified in Table 5-7. The Figure 5-102 details what Rock elements are built from the different abstractions, and which Mako templates are used for each element.

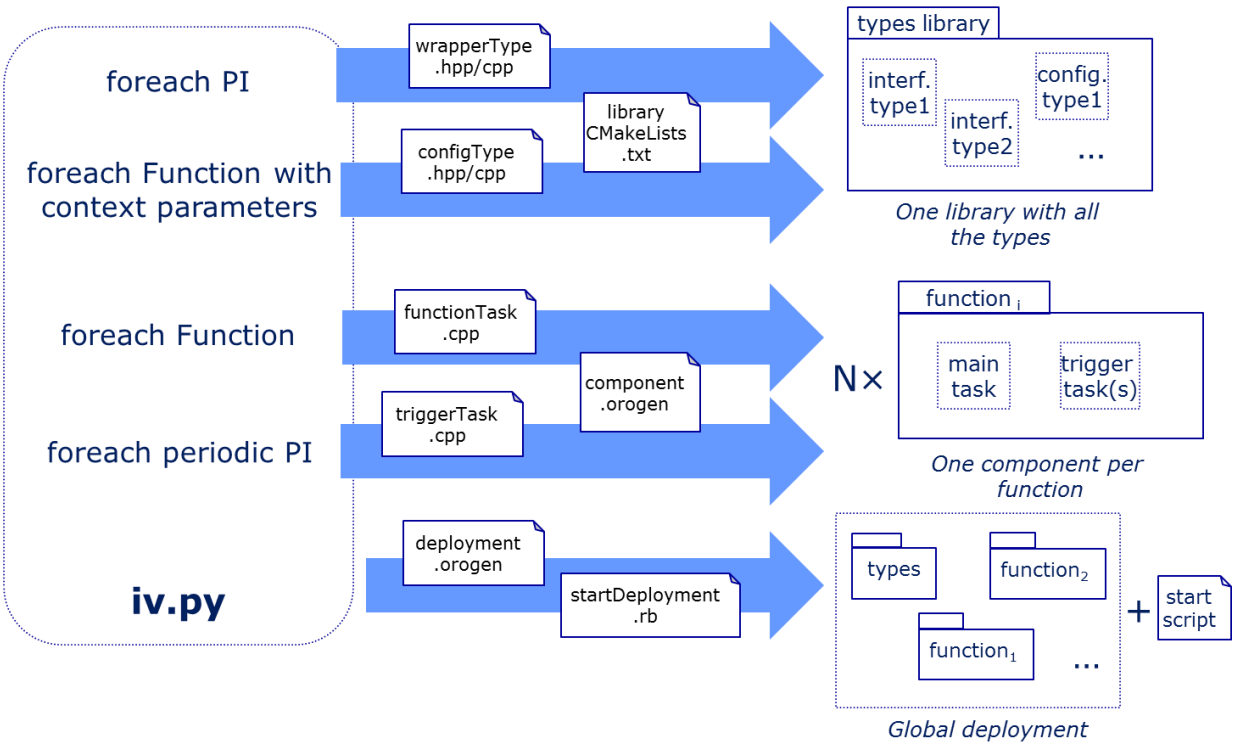


Figure 5-102. TASTE element mapping and template usage

A single **types library** is created for the TASTE model, containing:

- One type, based on the *wrapperType.hpp/cpp* templates, for each Provided Interface.
- One type, based on the *configType.hpp/cpp* templates, for the context parameters structure of each Function that defines context parameters.

The template *libraryCMakeLists.txt* is used to generate the build scripts of the types library.

One **function component** is created for each of the Functions in the TASTE model. The component specification is generated with the *component.orogen* template. The component contains:

- One main task, based on the template *functionTask.cpp*.
- An additional trigger task, based on the *triggerTask.cpp* template, for each Periodic Interface that the Function has.

A global **deployment** is created for the model, generated with the *deployment.orogen* template. This is a deployment in the Rock's sense, i.e. a set of tasks and libraries, and does not take into account the TASTE Deployment View. The deployment contains the types library and all the function components. In addition, a Ruby start script is created for launching the deployment, based on the *startDeployment.rb* template.

5.7. MANAGEMENT OF COMPONENT BUILD AND DEPENDENCIES

5.7.1.ARCHITECTURE OF THE DEVELOPMENT ENVIRONMENT

The development environment of ESROCOS is based on the idea of integrating various software components, which are developed and maintained individually, into one robotic system. Those components can stem from different sources and require different handling and may rely in turn on other sub-dependencies. The ESROCOS development tools themselves are not organized in a single software, but are made up in the same way, distributed over a set of independently maintained source code repositories.

To manage this architecture, the ESROCOS development tools rely on Autoproj, a tool to manage configurations of interdependent software sources. Autoproj acts as a means to check out, update, build and install software components as configured. Software packages are described by information regarding its build system, storage location and type. They are collected in so-called "package sets" which allow the organization of software packages. For ESROCOS a number of Package Sets have been created each of which describing a different set of software packages:

- The "core"-Package Set hold all libraries and tools, such as ESDEV, types or tools, libraries or components, which are considered as core features of ESROCOS.
- Within the "external"-Package Set, externally libraries or tools are collected, such as third party libraries or operating dependencies.
- A growing collection of ESROCOS compatible software components will be assembled in the "universe" package set. Here drivers for particular devices, or algorithms for data processing etc. will be collected.

With ESROCOS we want to establish a basis for collecting compatible software component from different vendors. To contribute to the overall software collection of ESROCOS eventually means listing a ESROCOS compatible component within the universe package set.

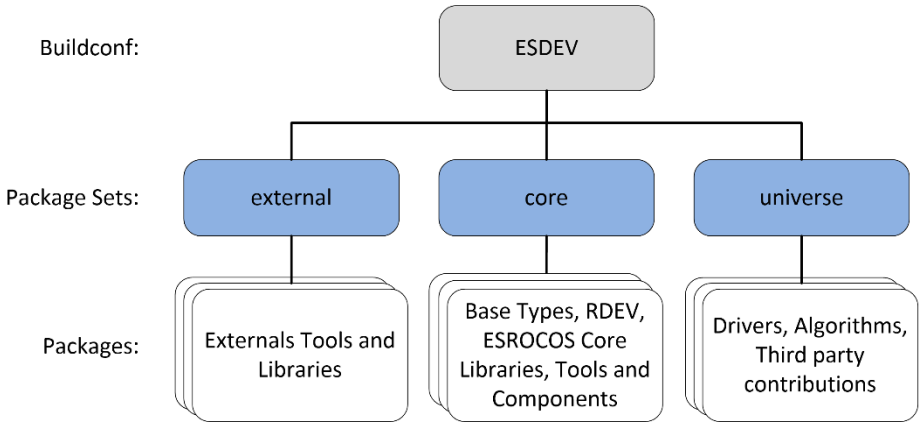


Figure 5-103. Overview of the ESROCOS base packages

As can be seen in Figure 5-103, the "buildconf" defines the layout of a ESROCOS software installation by importing the three ESROCOS main package sets, from which particular software packages are selected which are installed by the Autoproj tool.

5.7.2. ESROCOS DEVELOPMENT SCRIPTS

The ESROCOS development scripts are supposed to support developers in their work of developing ESROCOS compliant software components or systems. The play a similar role as the development scripts known from TASTE – in fact there a scripts with the same purpose for both, ESROCOS and plain TASTE. The difference is, that within ESROCOS the development paradigm is the collaborative multi-project development while plain TASTE is rather focused on a from scratch single-project development. As shown in Figure 5-104, the ESROCOS development scripts are often based on their TASTE-counterparts and in a sense play the role of a mediator between the multi-project world (Autoproj) and the component development world (TASTE).

Figure 5-104. ESROCOS Development Scripts

They are built to augment the original TASTE workflow with additional capabilities such as the management of project dependency configurations and enabling the later reuse of developed components as well as the integration of third party software. The following paragraphs will briefly explain the ESROCOS development scripts.

esrocos_create_project

In an interactive dialogue an ESROCOS project is initialized by calling esrocos_create_project which will create a project folder within the workspace as well as several TASTE source files such as the ASN.1/ACN files and default CMakeLists.txt and pkg-config template for handling dependencies. Package configuration attributes asked during the interactive dialog are stored in a esrocos.yml file which can be edited manually to accommodate for changes.

esrocos_fetch_dependencies

From the software dependency information in the esrocos.yml an autoproj manifest is generated to check out and install software packages the project depends on in the workspace. Subsequently required files are copied or linked to from within the project folder and ASN type files are added to the projects dataview.aadl. This step can be repeated at any stage of the development workflow to accommodate for changes in the project setup.

esrocos_edit_project

The software system or component to be developed is modelled within the TASTE GUI which is called with the esrocos_edit_project tool. The scripts controls the arguments passed to the TASTE GUI such that dependent components and types are imported. Just as in the development of a TASTE component, the software is modelled in the form of functions and interfaces. In the same way, the deployment modelling and software analysis tools are used. Importable components can be found in the project folder and used via the GUI tools.

esrocos_generate_skeletons

For the non-imported functions code stubs can be generated via esrocos_generate_skeletons. This function is mainly a helper to call the original TASTE tool but accommodates for differences in the file structure.

esrocos_build_project

This tool also calls the original TASTE build tool, but also utilizes Autoproj and CMake to collect linking information for any (sub) dependency that has to be provided to the actual build tool.

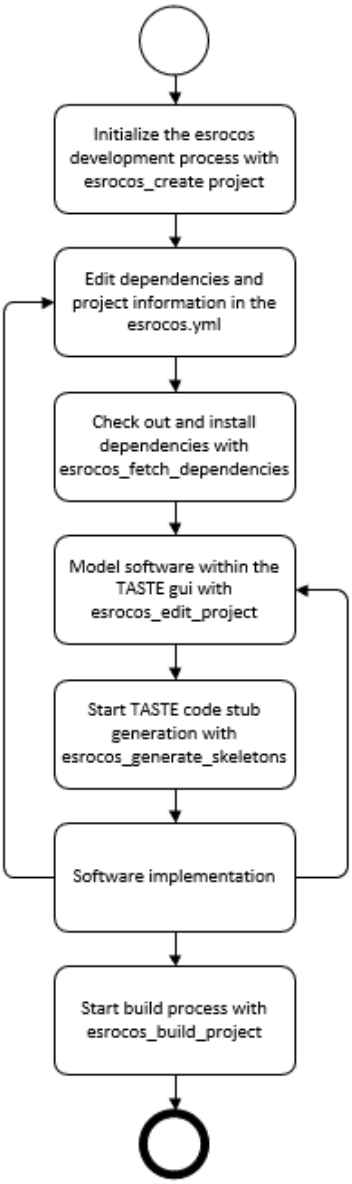


Figure 5-105. Development workflow guided by ESROCOS development scripts

END OF DOCUMENT

