ESROCOS
PRELIMINARY DESIGN DOCUMENT
ESROCOS_D2.2

<table>
<thead>
<tr>
<th>Due date of deliverable:</th>
<th>31-07-2017</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start date of project:</td>
<td>01-11-2016</td>
</tr>
<tr>
<td>Duration:</td>
<td>27 months</td>
</tr>
<tr>
<td>Topic:</td>
<td>COMPET-4-2016 Building Block a) Space Robot Control Operating System</td>
</tr>
<tr>
<td>Work package:</td>
<td>2100, 2200</td>
</tr>
<tr>
<td>Lead partner for this deliverable:</td>
<td>GMV</td>
</tr>
</tbody>
</table>

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

**Dissemination Level**

<table>
<thead>
<tr>
<th>PU</th>
<th>Public</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO-1</td>
<td>Confidential, restricted under conditions set out in Model Grant Agreement. Version providing the PSA will all the information required to perform its assessment.</td>
<td></td>
</tr>
<tr>
<td>CO-2</td>
<td>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</td>
<td></td>
</tr>
</tbody>
</table>

Prepared by: ESROCOS team

Approved by: GMV

Authorized by: Miguel Muñoz

Code: ESROCOS_D2.2

Version: 1.1

Date: 06/09/2017

Internal code: GMV 22280/17 V2/17

© ESROCOS Consortium 2017, all rights reserved
DOCUMENT STATUS SHEET

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Pages</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>23/06/2017</td>
<td>95</td>
<td>First issue of the document.</td>
</tr>
<tr>
<td>1.1</td>
<td>06/09/2017</td>
<td>95</td>
<td>Updates per PDR RIDs: sections 5.2.2 and 5.2.5.</td>
</tr>
</tbody>
</table>

NOTICE

The contents of this document are the copyright of the ESROCOS Consortium and shall not be copied in whole, in part or otherwise reproduced (whether by photographic, reprographic or any other method) and the contents thereof shall not be divulged to any other person or organisation without the prior written consent of the ESROCOS Consortium. Such consent is hereby automatically given to the European Commission and PERASPERA PSA to use and disseminate.
# TABLE OF CONTENTS

1. INTRODUCTION ........................................................................................................... 9  
   1.1. PURPOSE ................................................................................................................. 9  
   1.2. SCOPE .................................................................................................................... 9  
   1.3. CONTENTS ............................................................................................................... 10  

2. REFERENCE AND APPLICABLE DOCUMENTS .......................................................... 11  
   2.1. APPLICABLE DOCUMENTS .................................................................................... 11  
   2.2. REFERENCE DOCUMENTS ...................................................................................... 11  

3. TERMS DEFINITIONS AND ABBREVIATED TERMS .................................................. 13  
   3.1. DEFINITIONS ......................................................................................................... 13  
   3.2. ACRONYMS .......................................................................................................... 13  

4. SOFTWARE OVERVIEW .............................................................................................. 16  
   4.1. SCOPE AND WORKFLOW ...................................................................................... 16  
   4.2. SOFTWARE COMPONENTS IDENTIFICATION ...................................................... 17  

5. SOFTWARE COMPONENT DESIGN .............................................................................. 21  
   5.1. MODELLING OF KINEMATIC CHAINS ................................................................... 21  
   5.2. MODELING AND ANALYSIS OF DISTRIBUTED REAL-TIME SYSTEMS .......... 26  
      5.2.1. TASTE Improvements .................................................................................... 26  
      5.2.2. TASTE2BIP ..................................................................................................... 27  
         5.2.2.1. Use Case "Translate TASTE Models to BIP Models" .................................. 28  
         5.2.2.2. Use Case "Correct TASTE Model" ............................................................... 28  
         5.2.2.3. Translation Principles ................................................................................ 28  
         5.2.2.4. Design and Technologies ......................................................................... 29  
      5.2.3. BIP Compiler & Engine ................................................................................... 30  
         5.2.3.1. Use Case "Generate (C++) Code" ................................................................. 30  
         5.2.3.2. Use Case "Correct BIP Model" ................................................................. 31  
         5.2.3.3. Use Case "Run Simulation" ...................................................................... 31  
         5.2.3.4. Use Case "Run Real-Time Simulation" ..................................................... 31  
         5.2.3.5. Extensions ................................................................................................. 31  
         5.2.3.6. Design and Technologies ......................................................................... 32  
      5.2.4. SMC-BIP .......................................................................................................... 32  
         5.2.4.1. Theoretical background ............................................................................ 32  
         5.2.4.2. Use Case "Run SMC" .............................................................................. 33  
         5.2.4.3. Use Case "Run Qualitative SMC" ............................................................. 33  
         5.2.4.4. Use Case "Run Quantitative SMC" ............................................................ 34  
         5.2.4.5. Extensions ............................................................................................... 34  
         5.2.4.6. Design and Technologies ......................................................................... 34  
      5.2.5. FDIR Implementation and Analysis ................................................................. 36  
         5.2.5.1. FDIR analysis ............................................................................................. 36  
         5.2.5.2. FDIR implementation and integration ....................................................... 37  
         5.2.5.3. Use Case "Implement FDIR in TASTE" .................................................... 37
5.2.5.4. Use Case “Run V&V” ................................................................. 38
5.2.5.5. Use Case “Implement Faults Behaviour in BIP” ...................... 38
5.2.5.6. Use Case “Implement and Integrate BIP FDIR in TASTE Design” 38

5.3. COMMON ROBOTICS FUNCTIONS .................................................. 39
5.3.1. Base Robotics Data Types ........................................................... 39
5.3.2. OpenCV ......................................................................................... 47
5.3.3. Eigen ............................................................................................. 47
5.3.4. Transformer Library ...................................................................... 47
5.3.5. Stream Aligner Library .................................................................. 48
5.3.6. PUS Services ................................................................................ 48
5.3.6.1. Overview ................................................................................... 48
5.3.6.2. Services Identification ............................................................... 49
5.3.6.3. PUS_SERVICES Technical Overview ......................................... 50
5.3.6.4. PUS_API .................................................................................... 52
5.3.6.5. Event_Handler ........................................................................... 52
5.3.6.6. OBCP_Engine .......................................................................... 53
5.3.6.7. OBCP Programming Language ................................................. 54
5.3.6.8. DataPool_API ............................................................................ 56

5.4. DEPLOYMENT AND EXECUTION OF APPLICATIONS ..................... 57
5.4.1. AIR Hypervisor ............................................................................. 57
5.4.1.1. AIR Tool Chain Adaption to TASTE ........................................ 57
5.4.1.2. Upgrade of RTEMS .................................................................... 58
5.4.2. HAIR Emulator ............................................................................ 60
5.4.3. CAN Bus Driver ........................................................................... 61
5.4.4. Ethernet/EtherCAT Driver .............................................................. 62
5.4.5. SpaceWire Driver .......................................................................... 65

5.5. MONITORING, DEBUGGING AND TESTING ...................................... 67
5.5.1. Data Logger ................................................................................... 67
5.5.2. vizkit3d Integration ...................................................................... 67
5.5.3. Integration of ROS Assets ................................................................. 72
5.5.4. PUS Console ................................................................................ 72

5.6. INTEGRATION OF LEGACY SOFTWARE ............................................ 73
5.6.1. Middleware Bridges ...................................................................... 73
5.6.1.1. Description of the Approach ....................................................... 73
5.6.1.2. TASTE Components as Shared Library ....................................... 74
5.6.1.3. TASTE Components as Python Modules ...................................... 74
5.6.1.4. TASTE-ROCK Bridge ................................................................. 75
5.6.1.5. TASTE-ROS Bridge .................................................................. 77
5.6.2. Framework Import Tools ................................................................. 79
5.6.2.1. Import Tools for ROCK ............................................................... 79
5.6.2.2. Import Tools for ROS ................................................................. 81
5.6.3. Framework Export Tools ................................................................. 81
5.6.3.1. Export from TASTE to ROCK .................................................... 81
5.6.3.2. Export from TASTE to ROS ....................................................... 84
5.7. MANAGEMENT OF COMPONENT BUILD AND DEPENDENCIES .......................86
  5.7.1. Autoproj .............................................................................86
  5.7.2. ESROCOS Development Scripts ............................................88
6. TRACEABILITY BETWEEN REQUIREMENTS AND DESIGN.........................93
LIST OF TABLES AND FIGURES

Table 2-1. Applicable documents .................................................................11
Table 2-2. Reference documents .................................................................12
Table 3-1. Definitions .....................................................................................13
Table 3-2. Acronyms .....................................................................................13
Table 4-1. ESROCOS software components .................................................18
Table 5-1. Basic robotics data types ...............................................................45
Table 5-2. Requirements compliance .............................................................49
Table 5-3. Mapping between TASTE Interface View and Orocos elements ........76
Table 5-4. Mapping between TASTE Interface View and ROS elements ...........77
Table 5-5. ROS primitive types and their representation in C++ and Python .......78
Table 5-6. ROS array types and their representation in C++ and Python ..........78
Table 5-7. ROCK to TASTE elements mapping .............................................79
Table 5-8. Mapping of TASTE to ROCK concepts in taste2rock ....................82
Table 5-9. Mapping between TASTE Interface View and ROS elements ...........84
Table 6-1. Traceability between requirements and design ................................93

Figure 4-1. Development of a robot control application with ESROCOS ...........16
Figure 5-1. Overview of the Robot Modelling Tools .........................................21
Figure 5-2. Robotics modelling languages and relationships ............................22
Figure 5-3. Robot modelling workflow ..........................................................23
Figure 5-4. Software architecture of the robot modelling tool ...........................24
Figure 5-5. The ASSERT process ...................................................................26
Figure 5-6. Use Case diagram of the TASTE2BIP tool ...................................27
Figure 5-7. Workflow of TASTE2BIP ............................................................28
Figure 5-8. Translation of a TASTE cyclic PI sig as a BIP timed automaton ......29
Figure 5-9. Architecture of the TASTE2BIP tool ..........................................29
Figure 5-10. Use Case diagram of the BIP compiler and engine ......................30
Figure 5-11. BIP compiler and engine workflow and internal architecture ..........30
Figure 5-12. Use Case diagram of the SMC-BIP tool .....................................32
Figure 5-13. Workflow and architecture of SMC-BIP .....................................35
Figure 5-14. SMC-BIP GUI .........................................................................36
Figure 5-15. Use Case diagram for FDIR implementation and analysis ..........37
Figure 5-16. Timestamps and sensor data types .............................................40
Figure 5-17. Algebraic base types to describe SO(3) and SE(3) – part 1 ............41
Figure 5-18. Algebraic base types to describe SO(3) and SE(3) - part 2 ............42
Figure 5-19. Command base types to send control information to robots ..........42
Figure 5-20. Base types for data samples ................................................................. 43
Figure 5-21. Example for ASN.1’s Parametrized Types Feature .......................... 44
Figure 5-22. Template Types with AADL Pre-processor ........................................... 44
Figure 5-23. Type specification, installation and inclusion ....................................... 47
Figure 5-24. The space to ground PUS service system context .............................. 48
Figure 5-25. PUS_SERVICES component preliminary design .............................. 51
Figure 5-26. PUS_API preliminary design ............................................................... 52
Figure 5-27. Event_Handler preliminary design ......................................................... 53
Figure 5-28. OBCP_Engine preliminary design ......................................................... 54
Figure 5-29. DataPool preliminary design ............................................................... 56
Figure 5-30. AIR Tool Chain .................................................................................... 57
Figure 5-31. Comparison of RTEMS 4.8impr vs RTEMS 4.8 (start.S file) .............. 59
Figure 5-32. Comparison of RTEMS 4.8impr vs RTEMS 4.8 (start.S file) .............. 60
Figure 5-33. Class diagram for the PolyORB-HI CAN driver ................................ 61
Figure 5-34. Typical interface view used for CAN driver ........................................... 61
Figure 5-35. Deployment view specifying CAN driver and bus ............................... 62
Figure 5-36. Class diagram for the PolyORB-HI low-level Ethernet driver ........... 63
Figure 5-37. Typical TASTE Interface View used for Ethernet driver ..................... 64
Figure 5-38. TASTE Deployment View specifying Ethernet driver and bus ............ 64
Figure 5-39. Class diagram for the PolyORB-HI low-level SpaceWire driver ....... 65
Figure 5-40. Typical TASTE Interface View used for SpaceWire driver ............... 66
Figure 5-41. TASTE Deployment View specifying SpaceWire driver and bus .......... 66
Figure 5-42. vizkit3d plugin in TASTE IV (detail of sporadic PI) ......................... 68
Figure 5-43. vizkit3d window running in TASTE application ............................... 68
Figure 5-44. Component architecture of the vizkit3d-TASTE integration ........... 70
Figure 5-45. vizkit3d plugin functions in TASTE ...................................................... 71
Figure 5-46. PUS_CONSOLE preliminary design ..................................................... 72
Figure 5-47. TASTE-ROCK bridge ...................................................................... 73
Figure 5-48. TASTE-ROS bridge ........................................................................ 73
Figure 5-49. TASTE binary as shared library .......................................................... 74
Figure 5-50. Example of TASTE GUI ................................................................... 75
Figure 5-51. Communication with TASTE using a Python API ............................ 75
Figure 5-52. taste2rock transform logic ................................................................. 84
Figure 5-53. Package oriented software development ............................................ 86
Figure 5-54. Package oriented software development - Implementation workflow ... 87
Figure 5-55. Autoproj project specification ............................................................. 88
Figure 5-56. ESROCOS Component Development Workflow ............................... 90
Figure 5-57. ESROCOS System Integration Workflow ......................................... 91
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 Preliminary Design Document presents the overall architecture of the system and the software design of its main components.

1.2. SCOPE

This document is an outcome of the WP 2100 “RCOS Product Definition” and 2200 “Architecture Modelling” of the ESROCOS activity. These WPs establish the preliminary 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 identify the software components that constitute the framework, both for the RCOS and the RDEV, and outline their design. In the ESROCOS project some requirements will be 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.

This Preliminary Design Document reflects the state of the design at the PDR review. The design information will be completed later, and consolidated in the Detailed Design Document (D3.1), to be delivered at CDR.
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.
- Section 6: Traceability from Requirements to Design. Traces each of the ESROCOS software products to the requirements that it fulfils.
2. REFERENCE AND APPLICABLE DOCUMENTS

2.1. APPLICABLE DOCUMENTS

The following is the set of documents that are applicable:

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Title</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>[AD.1]</td>
<td>Peraspera: D3.1 Compendium of SRC Activities (for call 1)</td>
<td></td>
</tr>
<tr>
<td>[AD.2]</td>
<td>Guidelines for strategic research cluster on space robotics technologies horizon 2020 space call 2016</td>
<td></td>
</tr>
<tr>
<td>[AD.4]</td>
<td>ESROCOS System Requirements (D1.2) issue 1.1. GMV 2017/2/17 V2/17</td>
<td>23/06/2017</td>
</tr>
<tr>
<td>[AD.5]</td>
<td>ESROCOS Interface Control Document (D2.3) issue 2.0. GMV 2017/7/17 V2/17</td>
<td>23/06/2017</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Title</th>
<th>Date</th>
</tr>
</thead>
</table>

© ESROCOS Consortium 2017, all rights reserved

Preliminary Design Document
<table>
<thead>
<tr>
<th>Ref.</th>
<th>Title</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>[RD.24]</td>
<td>ROS tf. Online at <a href="http://wiki.ros.org/tf">http://wiki.ros.org/tf</a></td>
<td>-</td>
</tr>
<tr>
<td>[RD.26]</td>
<td>Collada. Online at: <a href="https://www.khronos.org/collada/">https://www.khronos.org/collada/</a></td>
<td>-</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Concept / Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.2. ACRONYMS

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

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AADL</td>
<td>Architecture Analysis and Design Language</td>
</tr>
<tr>
<td>AD</td>
<td>Applicable Document</td>
</tr>
<tr>
<td>AIR</td>
<td>ARINC 653 Interface in RTEMS</td>
</tr>
<tr>
<td>ANTLR</td>
<td>Another Tool for Language Recognition</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ARINC</td>
<td>Aeronautical Radio, Incorporated</td>
</tr>
<tr>
<td>ASN.1</td>
<td>Abstract Syntax Notation One</td>
</tr>
<tr>
<td>ASSERT</td>
<td>Automated proof-based System and Software Engineering for Real-Time applications</td>
</tr>
<tr>
<td>BIP</td>
<td>Behaviour, Interaction, Priority</td>
</tr>
<tr>
<td>BLTL</td>
<td>Bounded Linear Temporal Logic</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>CDR</td>
<td>Critical Design Review</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>CPDU</td>
<td>Command Pulse Distribution Unit</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DOF</td>
<td>Degrees of Freedom</td>
</tr>
<tr>
<td>ECSS</td>
<td>European Cooperation for Space Standardization</td>
</tr>
<tr>
<td>ERGO</td>
<td>European Robotic Goal-Oriented Autonomous Controller</td>
</tr>
<tr>
<td>ESA</td>
<td>European Space Agency</td>
</tr>
<tr>
<td>ESROCOS</td>
<td>European Space Robotics Control and Operating System</td>
</tr>
<tr>
<td>FDIR</td>
<td>Failure Detection, Isolation and Recovery</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In, First-Out</td>
</tr>
<tr>
<td>GCC</td>
<td>GNC Compiler Collection</td>
</tr>
<tr>
<td>GPL</td>
<td>General Public License</td>
</tr>
<tr>
<td>GR</td>
<td>Gaisler Research</td>
</tr>
<tr>
<td>GS</td>
<td>Ground Segment</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HAIR</td>
<td>Hypervisor emulator based on AIR</td>
</tr>
<tr>
<td>Acronym</td>
<td>Definition</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Description Language</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IMU</td>
<td>Inertial Measurement Unit</td>
</tr>
<tr>
<td>IO</td>
<td>Input/Output</td>
</tr>
<tr>
<td>ITU-T</td>
<td>ITU Telecommunication Standardization Sector</td>
</tr>
<tr>
<td>IV</td>
<td>Interface View</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>LTS</td>
<td>Long-Term Support</td>
</tr>
<tr>
<td>OBCP</td>
<td>On-Board Control Procedure</td>
</tr>
<tr>
<td>OROCOS</td>
<td>The Open Robot Control Software</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OSG</td>
<td>Open Scene Graph</td>
</tr>
<tr>
<td>PBLTL</td>
<td>Probabilistic Bounded Linear Temporal Logic</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>PDD</td>
<td>Preliminary Design Document</td>
</tr>
<tr>
<td>PDR</td>
<td>Preliminary Design Review</td>
</tr>
<tr>
<td>PI</td>
<td>Provided Interface</td>
</tr>
<tr>
<td>POS</td>
<td>Partition Operating System</td>
</tr>
<tr>
<td>POSIX</td>
<td>Portable Operating System Interface Unix</td>
</tr>
<tr>
<td>PSA</td>
<td>Programme Support Activity</td>
</tr>
<tr>
<td>PUS</td>
<td>Packet Utilization Standard</td>
</tr>
<tr>
<td>PWM</td>
<td>Pulse Width Modulation</td>
</tr>
<tr>
<td>Q&amp;A</td>
<td>Questions &amp; Answers</td>
</tr>
<tr>
<td>QUDT</td>
<td>Quantities, Units, Dimensions and Types</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RAMS</td>
<td>Reliability, Availability, Maintainability and Safety</td>
</tr>
<tr>
<td>RCOS</td>
<td>Robot Control Operating System</td>
</tr>
<tr>
<td>RD</td>
<td>Reference Document</td>
</tr>
<tr>
<td>RDEV</td>
<td>RCOS Development Environment</td>
</tr>
<tr>
<td>RI</td>
<td>Required Interface</td>
</tr>
<tr>
<td>ROCK</td>
<td>Robot Construction Kit</td>
</tr>
<tr>
<td>ROM</td>
<td>Read-Only Memory</td>
</tr>
<tr>
<td>ROS</td>
<td>Robot Operating System</td>
</tr>
<tr>
<td>RTEMS</td>
<td>Real-Time Executive for Multiprocessor Systems</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
<tr>
<td>RTT</td>
<td>Real-Time Toolkit</td>
</tr>
<tr>
<td>SARGON</td>
<td>Space Automation and Robotics General Controller</td>
</tr>
<tr>
<td>SDL</td>
<td>Specification and Description Language</td>
</tr>
<tr>
<td>SLAM</td>
<td>Simultaneous Localisation and Mapping</td>
</tr>
<tr>
<td>SMC</td>
<td>Statistical Model Checking</td>
</tr>
<tr>
<td>Acronym</td>
<td>Definition</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>SPARC</td>
<td>Scalable Processor Architecture</td>
</tr>
<tr>
<td>SPRT</td>
<td>Sequential Ratio Testing Procedure</td>
</tr>
<tr>
<td>SRC</td>
<td>Strategic Research Cluster</td>
</tr>
<tr>
<td>SRR</td>
<td>System Requirements Review</td>
</tr>
<tr>
<td>SSP</td>
<td>Single Sampling Plan</td>
</tr>
<tr>
<td>STL</td>
<td>Standard Template Library</td>
</tr>
<tr>
<td>SVN</td>
<td>Subversion</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>TASTE</td>
<td>The ASSERT Set of Tools for Engineering</td>
</tr>
<tr>
<td>TASTE-CV</td>
<td>TASTE Concurrency View</td>
</tr>
<tr>
<td>TBC</td>
<td>To Be Confirmed</td>
</tr>
<tr>
<td>TBD</td>
<td>To Be Determined</td>
</tr>
<tr>
<td>TC</td>
<td>Telecommand</td>
</tr>
<tr>
<td>TM</td>
<td>Telemetry</td>
</tr>
<tr>
<td>TRL</td>
<td>Technology Readiness Level</td>
</tr>
<tr>
<td>TSP</td>
<td>Time and Space Partitioning</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>URDF</td>
<td>Unified Robot Description Format</td>
</tr>
<tr>
<td>V&amp;V</td>
<td>Verification &amp; Validation</td>
</tr>
<tr>
<td>VCS</td>
<td>Version Control System</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst-Case Execution Time</td>
</tr>
<tr>
<td>WP</td>
<td>Work Package</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Mark-up Language</td>
</tr>
<tr>
<td>YAML</td>
<td>YAML Ain't Mark-up Language</td>
</tr>
</tbody>
</table>
4. SOFTWARE OVERVIEW

4.1. SCOPE AND WORKFLOW

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.

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 4-1 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 4-1. 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 in 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.

### 4.2. SOFTWARE COMPONENTS IDENTIFICATION

The components of the ESROCOS framework are defined according to the activities supported by the framework and shown in Figure 4-1.

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.

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. Three categories are established:

- New development: the component is fully developed within the project (either from scratch or based on prior work), and a complete design is provided.
- Extension and integration of existing SW: an existing software component is extended with new capabilities or significantly reworked to fulfil the requirements of ESROCOS, and integrated into the framework. The design information provided covers the improvements of the component and its integration in ESROCOS.
- Integration of existing SW: an existing software component, possibly from a third party, is integrated in ESROCOS with minor or no modification. The design information provided covers its integration in ESROCOS.

The Table 4-1 enumerates the components of the ESROCOS framework, classified according to their type (RCOS or RDEV), quality level (laboratory or space) and scope of the work required (new development, extension or just integration).

<table>
<thead>
<tr>
<th>Activity</th>
<th>Component</th>
<th>RCOS</th>
<th>RDEV</th>
<th>Lab</th>
<th>Scope of the work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model kinematic chains</td>
<td>Robot modelling tools</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>New development</td>
</tr>
<tr>
<td>Model and analyse distributed real-time systems</td>
<td>TASTE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Extension and integration of existing SW</td>
</tr>
<tr>
<td>BIP compiler</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>Extension and integration of existing SW</td>
</tr>
<tr>
<td>BIP engine</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>Extension and integration of existing SW</td>
</tr>
<tr>
<td>TASTE2BIP</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>New development</td>
</tr>
<tr>
<td>SMC-BIP</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>Extension and integration of existing SW</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solvers for all possible kinematics and dynamics transformations of lumped parameter robot chains are generated from formal and semantically validatable models.</td>
<td></td>
</tr>
<tr>
<td>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</td>
<td></td>
</tr>
<tr>
<td>Compiler tool for generating C++ code from BIP models.</td>
<td></td>
</tr>
<tr>
<td>Runtime for executing C++ code generated from BIP models.</td>
<td></td>
</tr>
<tr>
<td>Generation of BIP models from TASTE models.</td>
<td></td>
</tr>
<tr>
<td>Statistical model-checker for BIP models.</td>
<td></td>
</tr>
<tr>
<td>Activity</td>
<td>Component</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Common robotics functions</td>
<td>Base robotics data types</td>
</tr>
<tr>
<td></td>
<td>OpenCV</td>
</tr>
<tr>
<td></td>
<td>Eigen</td>
</tr>
<tr>
<td></td>
<td>Transformer</td>
</tr>
<tr>
<td></td>
<td>Stream aligner</td>
</tr>
<tr>
<td></td>
<td>PUS services</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Deploy and run</td>
<td>AIR</td>
</tr>
<tr>
<td></td>
<td>HAIR</td>
</tr>
<tr>
<td></td>
<td>CAN bus driver</td>
</tr>
<tr>
<td></td>
<td>Ethernet driver</td>
</tr>
<tr>
<td></td>
<td>SpaceWire driver</td>
</tr>
<tr>
<td></td>
<td>EtherCAT driver</td>
</tr>
<tr>
<td>Monitor, debug, test</td>
<td>Data logger</td>
</tr>
<tr>
<td></td>
<td>vizKit3d</td>
</tr>
<tr>
<td></td>
<td>RVIZ</td>
</tr>
<tr>
<td></td>
<td>Gazebo</td>
</tr>
<tr>
<td></td>
<td>PUS console</td>
</tr>
<tr>
<td>Integrate legacy SW</td>
<td>Middleware bridges</td>
</tr>
<tr>
<td></td>
<td>Framework import</td>
</tr>
<tr>
<td></td>
<td>Framework export</td>
</tr>
</tbody>
</table>
## Activity

<table>
<thead>
<tr>
<th>Activity</th>
<th>Component</th>
<th>RCOS</th>
<th>RDEV</th>
<th>Lab</th>
<th>Space</th>
<th>Description</th>
<th>Scope of the work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manage build and dependencies</td>
<td>Autoproj</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>Software package management and build tool.</td>
<td>Integration of existing SW</td>
</tr>
<tr>
<td>ESROCOS development scripts</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>Collection of development tools for setting up/editing projects in the ESROCOS environment.</td>
<td>Extension and integration of existing SW</td>
</tr>
</tbody>
</table>

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

Modelling a kinematic chain is envisaged to work as follows: the developer imports an already existing model, that contains already a specific subset of the different modelling languages described below. In the editor, the developer can than change the model, or add new semantics, while the tool is providing information about the consistency and/or completeness of the model. For example, it will check whether physical units are added, and if so, whether they are consistent in sub-parts of the whole model, etc. When the model is complete, the developer will be able to identify which kind of “queries” he/she needs to have in the eventual software component; for example, the forward position kinematics is needed not just for the end-effector frame but also for an intermediate frame where a sensor is attached to the kinematic chain. With this combined inputs (model + queries), code is being generated that can be used in TASTE components, or for which stand-alone unit tests can be performed.

Figure 5-1. Overview of the Robot Modelling Tools

The scope of this activity is looking beyond the horizon of the current project, and provides a future-proof basis, to which, in follow-up projects, formal reasoning and model-based verification can be added. The result of this approach is a large set of rather small modelling languages, designed to be composed together in more ways than needed in the project. More concretely, the following set of modelling languages is foreseen:

- **Geometry**: primitives (e.g., point, line, frame, plane) and relationships (e.g., composition, distance, orthogonality, perpendicularity).

- **Mathematical representations**: primitives (e.g., three-vector, homogeneous transformation matrix, rotation matrix, quaternion, list) and relationships (e.g., composition, transformation, orthogonality constraint).

- **Numerical representations**: primitives (e.g., integer, float, array) and relationships (e.g., casting, boundary checking, data structures).

- **Digital representations**: primitives (e.g., number of bits in IEEE Float, bit alignment in C structs) and relationships (bit alignment constraint).

- **Uncertainties**: primitives (probability density functions) and relationships (conditional probabilities, normalization, Bayes’ rule).

- **Physical units**: more in particular, the “QUDT” ontology of Quantities, Units, Dimensions and Types as primitives, with transformation relationships between them.
- **Kinematic chains**: with link and joint as major primitives, and motion constraints and kinematic/dynamic transformations as major relationships.

- **Rigid bodies**: as the primitive to link all of the above together, in the context of robotics. The following “attachments” are essential aspects of the compositions that run best via the “Rigid body” concept, because that is the natural concept to add “attachment points” to.

- **Sensor attachments**: locations on Rigid bodies on Kinematic chains where sensors of various nature can be attached.

- **Mechanical domain**: inertia, damping/friction, elasticity/stiffness, transmission, play. All of them are attached somewhere on the Rigid body links in Kinematic chains.

- **Actuator domain**: electrical, hydraulic, pneumatic, etc., to be attached to Rigid Bodies on Kinematic chains.

- **Motion representations**: there are various families of motions, which will be ordered according to the following dependency order: only kinematic chain dependencies (with the “hybrid dynamics” algorithm being the most generic motion “solver”, taking into account the constraints imposed by the robot kinematics and actuation only), target object dependencies (“move to” constraints on the hybrid dynamics, where the constraints come only from one single target object), environment object dependencies (giving rise to “move constrained” type of motion specifications, because the robot gets extra constraints coming from objects in the environment that are not the target of the task).

The Figure 5-2 summarizes these modelling languages and the relationships among them. These relationships represent dependencies, in the sense that adding one “higher-level” primitive (e.g., a new “Joint”) to a model of a kinematic chain, automatically triggers the tool to help the developer make consistent and complete choices of mathematical representations, and physical units, etc.

---

**Figure 5-2. Robotics modelling languages and relationships**

From the *architectural* point of view, the robot modelling tool will provide software component (sub)architectures that the TASTE tool can process to make a concrete robot execute a concrete motion. The semantic richness of these software components will comprise “straight line” and “circular arc”, together with some more flexible “guarded motion” types, with tolerances on the trajectory; for example, to move a gripper parallel to a plane.

The formal models of the different kinematics and dynamics aspects of the robot are semantically validated and composed into an internal graph-based representation that
can be queried. At runtime, the robot control application may perform queries, e.g., to plan or execute a concrete motion. The tool generates the necessary code to perform such queries, as well as unit tests. To integrate the software in TASTE models, ASN.1 and AADL definitions are generated. This workflow is summarized in Figure 5-3.

**Figure 5-3. Robot modelling workflow**

The modelling of kinematic chains within the scope of the project is envisaged to be a workflow that supports the configuration of an enumerated and agreed upon set of "kinematic families" that are relevant. The two top-level families will be (1) robot arms with N serially connected one degree-of-freedom joints, and (2) mobile platforms with N wheel units connected in parallel to one car body. Each such family can be given sub-families, each of which makes a particular choice of some of the many topological and geometric parameters present in each family; for example, the common cases of serial robots with six revolute joints, or mobile platforms with six rocker-supported wheels.

At the level of the TASTE tool, the modelling of kinematic chains will be visible in two ways: in the Data View and in the Interface View. The former means that one set of Data Views in ASN.1 format is available for each kinematic family, formally representing all the data structures that can be needed in the Interface View: representation of all "geometric primitives" (points, vectors and frames) in the kinematic model of a particular Kinematic Family, and of all "motion states" of those "geometric primitives" (Cartesian position, speed and acceleration, joint position, speed and acceleration, joint torques) that can occur during a motion of the kinematic chain.

From a library perspective, it does not make a difference if the motion state representation is needed in the context of motion planning, or motion control, or motion estimation; all three of these processes will be present in all robotics systems, in different forms, but all this requires at the TASTE Data View level is to extend the kinematic ASN.1 data types with the appropriate meta data tags. Which tags are needed depends on which development processes will actually be implemented.

The activity will extend the Interface View of the TASTE Tool with function definitions for all functionalities offered by the kinematics library. That is, the forward and inverse velocity/force kinematics and dynamics, which are functions that take the above-mentioned ASN.1 data types (of geometric primitives and motion states) as inputs and outputs. The exact number of the provided functions depends on the needs of kinematic transformations in the project. The ESROCOS framework provides a C-library that can generate an implementation for any such function, on the basis of the following inputs: model of a kinematic chain, selection of which geometric primitives are involved in the requested kinematic transformation function, selection of the motion state that is requested as output.
There are two options for integration into TASTE: the above-mentioned interactive specification of new kinematic solver functions becomes part of the TASTE deployment, or it is done in an external stand-alone tool, and only the results are to be integrated into the Data and Interface Views in a static manner.

For all of the above data structures and functions, documentation will be available about the following "semantic" information that users of the tool and the library need to have access to: the choice of mathematical representation of the geometric and motion state primitives, the numerical representation, and the chosen physical units. For example, the numerical values in an ASN.1 data type of the type "frame velocity" represent the six-dimensional twist (that is, instantaneous velocity) with identified velocity reference points on the two bodies whose relative motion is represented; the first three float numbers represent the linear part of the velocity twist, in meters per seconds, and the last three float numbers represent the angular part of the velocity twist, in radians per seconds.

The "solver" library will be accompanied by an "import" library, that can generate and initialise the above-mentioned ASN.1 Data Types for, both, the kinematic family and kinematic chain parts described by an external file. A target format of this input file is Collada [RD.25][RD.26], an XML-based schema used to transport 3D assets between applications.

In particular, Collada format allows to store information about kinematic chains and their physical properties, which can be initially defined by an external program. Export to Collada cannot be guaranteed to be "lossless", since the Collada format does not have support for all the semantic properties that the Robot Modelling libraries support.

---

**Figure 5-4. Software architecture of the robot modelling tool**

The software architecture of the tool is presented in Figure 5-4. Its main components are detailed below:

- **Models Import**: load the models (of both a kinematic chain and the functional queries required to have available at runtime), and composes them into one integrated “robot model”.
- **Query Parser**: interprets the queries into a suitable internal representation.
- **Query Analyser**: performs semantic checking and validation, and possible returns information about semantic inconsistencies.

- **Composer**: manages the graph-based robot description (e.g., to add or remove valid properties)

- **Inference engine**: traverses the graph to answer the query requests

- **Code generation**: this is a family of queries (which the developer must not define himself, but that are “built-in” into the tool) that, together, generate a solver library for use by applications. It is composed of:
  - **TASTE/AADL generator**: generates the necessary models and source code to integrate the robot model component in TASTE models.
  - **Kin/Dyn Solver generator**: generates the actual solver implementation.
  - **ASN.1 compiler**: generates C code for the ASN.1 types used by the solver (the compiler is part of the core TASTE infrastructure).
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 5-5. 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 5-5 recapitulates 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, Xenomai;
- Proper integration and testing of drivers done by partners, validation they meet Q&A criteria depending on Space or Lab quality level;
- Supporting ESROCONS and other OG in their usage of TASTE, and addressing bug reports.

In addition, ISAE will prepare 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.

The TASTE toolset will also be extended to support the integration mechanisms between TASTE and external middleware as discussed in section 5.6.1.

5.2.2. TASTE2BIP

This new tool aims to translate a TASTE model into a BIP model. More precisely, we consider here from a TASTE model the data view, the interface view together with the software function behaviour in SDL, the deployment view and the concurrency view. The tool produces the corresponding BIP model. The uses cases of this tool are illustrated in Figure 5-6, and the tool workflow in Figure 5-7.

The TASTE2BIP tool is limited to software behaviour description in SDL. The only other programming language supported by TASTE that the tool will partially cover is C/C++. Code written in the latter will be taken into account in the translation as long as it is standalone, i.e., all computations are local to the component and no calls to functionalities implemented by other components are present. This feature is supported by the BIP language and tools which allows to include such standalone C/C++ code. Other modelling/programming language supported by TASTE such as Simulink, Ada, etc., will not be covered by the proposed implementation.

Figure 5-6. Use Case diagram of the TASTE2BIP tool
5.2.2.1. USE CASE “TRANSLATE TASTE MODELS TO BIP MODELS”

Description: This use case describes the main functionality of the tool, the model transformation of TASTE designs into BIP models.

Actor: The developer.

Input: A TASTE design consisting of data view, interface view, behaviour as SDL state machines and/or leaf C code, deployment view and concurrency view.

Output: A BIP model.

Basic flow: A TASTE design is translated in a BIP model if no modelling errors have been detected during this phase (e.g., a software function that has no interface).

Alternate flow: Modelling errors are detected at translation, then the tool stops and outputs them to the developer.

5.2.2.2. USE CASE “CORRECT TASTE MODEL”

Description: This use case covers the correction of a TASTE model in case it contains modelling errors. It is an extension of the “Translate TASTE models to BIP models” if the latter does not provide the expected result.

Actor: The developer.

Precondition: A list of modelling errors has been provided by the TASTE2BIP tool.

Input: A TASTE design and a list of modelling errors.

Output: The corrected TASTE design.

Basic flow: The provided modelling errors are inspected on the TASTE design and corrected accordingly such that a new design is created.

5.2.2.3. TRANSLATION PRINCIPLES

The main ideas of the TASTE to BIP translation are the following:

- The data types are represented with BIP basic types (bool, real, string) and data structures are mapped to similar counterparts.
- Every component of the interface view becomes a BIP timed automaton.
The SDL state machine of a component gives the behaviour of the timed automaton.

Every provided interface of a component becomes a timed automaton, where its type – cyclic/sporadic – indicates the behaviour pattern. As example is provided in Figure 5-8.

Time properties period/miniminal inter-arrival time and worse-case execution time (WCET) are taken into account during translation as execution assumptions; time property deadline is considered a requirement and will be used during verification and validation.

Interaction between the different components is described by signal name matching.

The deployment and concurrency views are taken into account during the refinement of the abstract BIP model into a physical (software/hardware) one.

Figure 5-8. Translation of a TASTE cyclic PI sig as a BIP timed automaton

5.2.2.4. DESIGN AND TECHNOLOGIES

The proposed architecture of TASTE2BIP is given in Figure 5-9. The main program is part of the package General and relies on two components: the Translator and the Parser. The Translator aims at generating for each TASTE element the corresponding BIP code. This package consists of a collection of classes, where each class translates a specific TASTE modelling element. Helper functions with respect to this functionality are defined in the Utilities package. The Parser aims at providing the basic modelling elements from a textual TASTE description (e.g., AADL code for the interface view). We distinguish here the need for 3 parser components: for data types the ASN parser, for interface, deployment and concurrency view the TASTE Parser, and for behaviour the SDL Parser. Each of these parsers is a component from the TASTE framework, and will not be developed separately. For example, for interface views we will use the Ocarina tool from TASTE which contains an AADL parser.

The TASTE2BIP tool will be implemented in Python. This choice is supported by the third-party parsers intended to be used in the development and which provide Python bridges for easier coding.

Figure 5-9. Architecture of the TASTE2BIP tool
5.2.3. BIP COMPILER & ENGINE

The BIP compiler and engine are the core tools supporting the BIP language. BIP stands for Behaviour, Interaction, and Priority. The BIP language is:

- Model-based: its semantics is formally defined in terms of operational semantic rules
- Component-based with a clean notion of component interface from which composite components can be built

BIP models are hierarchically structured from atomic components, which are characterised by their behaviour and interfaces. This allows developers to compose components by layered application of interactions and priorities. Architecture is a first-class concept in BIP, with well-defined semantics that system designers can analyse and perform.

The BIP compiler and engine allow to generate code (in C++ for example) and to execute this code as different simulation types. The uses cases of these tools are illustrated in Figure 5-10. The tools workflow is provided in Figure 5-11.

![Figure 5-10. Use Case diagram of the BIP compiler and engine](image)

![Figure 5-11. BIP compiler and engine workflow and internal architecture](image)

5.2.3.1. USE CASE "GENERATE (C++) CODE"

Description: This use case covers the generation of (C++) code from a BIP model.

Actor: The developer.

Input: A BIP model.

Output: (C++) code.

Basic flow: The BIP model is initially parsed and static analysis is performed. The latter covers grammar compliance, typing analysis, etc. Then, via several model transformations, code is generated. This code is mainly distributed C++ code, but it can be generated in other programming languages as extension.

Alternate flow: The statistical analysis phase detected modelling errors in the model. The compiler stops and outputs the list of identified errors.
5.2.3.2. USE CASE "CORRECT BIP MODEL"

Description: This use case models the correction of a BIP model if errors have been identified during code generation. Therefore, this case is an extension of the "Generate (C++) code" use case.

Actor: The developer.

Precondition: A list of modelling errors has been provided to the user by the compiler.

Input: A BIP model and a list of modelling errors.

Output: The corrected BIP model.

Basic flow: The developer inspects the BIP model and the error list and provides a correction for each of them.

5.2.3.3. USE CASE "RUN SIMULATION"

Description: This use case covers the simulation functionality of the BIP engine.

Actor: The developer.

Input: The C++ code generated from a BIP model.

Output: A simulation trace, printed on screen.

Basic flow: The developer runs the engine with the desired compiled BIP model. The engine selects a possible execution and prints it to the user. This execution is a untimed basic one.

5.2.3.4. USE CASE "RUN REAL-TIME SIMULATION"

Description: This use case extends the simulation functionality by taking into consideration when outputting the execution the timing constraints.

Actor: The developer.

Input: The C++ code generated from a BIP model, and an option for real-time simulation.

Output: A real-time simulation trace, printed on screen.

Basic flow: The developer runs the engine with the desired compiled BIP model and the real-time option. The engine selects an execution and prints it to the user, while it respects all timing constraints (e.g., if a delay of 3s is the BIP model, the engine will wait for 3s between two output events).

5.2.3.5. EXTENSIONS

In the context of ESROCOS, we plan to extend the BIP compiler and the BIP engine(s) to express timing constraints, stochastic behaviour, and error models.

The (real-)time extension of the compilation chain will consider timed automata instead of standard automata for implementing the behaviour of components, that is, components may declare clocks, and guards on transitions may include constraints on clocks. Moreover, the BIP engine(s) will be extended to include two modes of execution: a simulation mode in which timing constraints are interpreted logically (time is simulated), and a real-time mode in which models are executed in real-time.

The compilation chain will also be extended to include stochastic attributes specifying probability distributions used for executing times transitions. Such an extension is necessary for building quantitative models and estimating performance metrics using statistical model-checking.
Lastly, we plan to extend the language to distinguish between normal behaviour and faults behaviour. This extension will be used for specifying error models in the FDIR activity.

### 5.2.3.6. DESIGN AND TECHNOLOGIES

The BIP compiler is organized in Java packages in a modular way, allowing the dynamic invocation of model-to-model transformers and code generators. It can be used to obtain executable code from BIP specifications which can be considered as executable models. We are mainly targeting C++ code, although other code generators and model-to-model transformers can be developed and integrated seamlessly in the compilation process. The internal architecture of the tool is illustrated in Figure 5-11.

The execution of the generated C++ code requires an execution engine which is a generic C++ library responsible for enforcing at runtime the semantics of interactions and priorities, and which is linked during the compilation process of the generated code. The BIP Engine has several modes of execution: simulation, real-time execution, exhaustive exploration of the reachable states (model-checking).

Installation instructions can be found at [http://www-verimag.imag.fr/New-BIP-tools.html](http://www-verimag.imag.fr/New-BIP-tools.html). The BIP compiler and engines are provided as an archive containing the binaries needed for executing the tool. In the course of the project, the installation will be integrated with the rest of the ESROCOS framework. The target platforms are GNU/Linux x86 based machines, however, the tool is known to work correctly on Mac OSX, and probably other Unix-based systems. The tool requires a Java VM (version 6 or above), a C++ compiler (preferably GCC) with the STL library, and the CMake build tool. More details are available on the same page, a detailed BIP documentation is available at [http://www-verimag.imag.fr/TOOLS/DCS/bip/doc/latest/html/index.html](http://www-verimag.imag.fr/TOOLS/DCS/bip/doc/latest/html/index.html).

### 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 of a system. SMC can be seen as an improvement of purely simulation-based techniques since it guarantees results with respect to a user-defined level of confidence. This is obtained in a SMC model-checker by computing a number of executing sequences sufficient to get the coverage of the system behaviour required by the level of confidence. The SMC basic concepts are described below, while the tool use cases are illustrated in Figure 5-12.

![Figure 5-12. Use Case diagram of the SMC-BIP tool](image)

#### 5.2.4.1. THEORETICAL BACKGROUND

In the following $B$ denotes a stochastic system and $\Phi$ a property of interest expressed in BLTL (Bounded Linear Temporal Logic). Statistical model-checking refers to a series of simulation-based techniques that can be used to answer two questions: (1) **Qualitative:** is the probability for $B$ to satisfy $\Phi$ greater or equal to a certain threshold $\theta$ and (2)
Quantitative: what is the probability for $B$ to satisfy $\Phi$? Both questions can serve to decide a PBLTL property.

The main approaches proposed to answer the qualitative question are based on hypothesis testing. Let $p$ be the probability of satisfaction of $\Phi$, to determine whether $p \leq \theta$, 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 ($\alpha, \beta$) of a test is determined by two parameters, $\alpha$ and $\beta$, such that the probability of accepting $K$ (respectively, $H$) when $H$ (respectively, $K$) holds is less or equal to $\alpha$ (respectively, $\beta$). 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 in $\theta$ 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. A logarithmic based algorithm has also been proposed, that given $p_0, p_1, \alpha$ and $\beta$ implements the Sequential Ratio Testing Procedure (SPRT). When one has to test $\theta \geq 1$ or $\Phi \geq 0$, it is however better to use Single Sampling Plan (SSP) that is another algorithm whose 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.

There also exist estimation procedures ($PESTIMATION$) to compute the probability $p$ for $B$ to satisfy $\Phi$. Given a precision $\delta$, such procedures compute a value for $p'$ such that $|p' - p| \leq \delta$ with confidence $1-\alpha$.

The efficiency of the above algorithms is characterised by the number of simulations needed to obtain an answer. This number may change from system to system and can only be estimated. However, some generalities are known. For the qualitative case, it is known that, except for some situations, SPRT is always faster than SSP. $PESTIMATION$ can also be used to solve the qualitative problem, but it is always slower than SSP. If $\theta$ is unknown, then a good strategy is to estimate it using $PESTIMATION$ with a low confidence and then validate the result with SPRT and a strong confidence.

5.2.4.2. USE CASE "RUN SMC"

Description: This use case describes the execution of the SMC module. The SMC relies of the execution of the basic BIP engine simulator (therefore, the inclusion relation).

Actor: The developer.

Input: The generated code of a BIP model, the requirement to verify, and a list of probabilities-related parameters.

Output: Specialized based on the type of SMC run.

Basic flow: The tools takes the BIP model and runs several simulations. The statistics are computed accordingly to the required functionality.

5.2.4.3. USE CASE "RUN QUALITATIVE SMC"

Description: This use case covers the qualitative aspect of SMC, i.e., whether the probability to satisfy a given requirement is greater or equal than a threshold.

Actor: The developer.

Input: The generated code of a BIP model, the requirement to check, the threshold, and the hypothesis testing parameters.

Output: A verdict if the hypothesis testing passes or fails.

Basic flow: Based on the statistics computed after the simulations (see sections 5.2.4.1 and 5.2.4.2), a result is provided to the user.
5.2.4.4. USE CASE "RUN QUANTITATIVE SMC"

Description: This use case covers the quantitative aspect of SMC, i.e., what is the probability of a model to satisfy a requirement.

Actor: The developer.

Input: The generated code of a BIP model, the requirement to check, and the precision.

Output: The computed probability estimation.

Basic flow: Based on the statistics computed after the simulation (see sections 5.2.4.1 and 5.2.4.2), an estimation probability is computed and outputted.

5.2.4.5. EXTENSIONS

We plan to extend the tool to real-time BIP specifications. Such an extension includes modifying the SMC core as well the real-time BIP engine to include a proper stochastic behaviour. However, the architecture of the tool and its main interfaces will remain unchanged.

Also, we plan to extended the logic used for formalizing requirements. In consequence, the user will be able to express richer requirements of the type: what is the probability for an event to occur within a given time-bound (after an event) and is the probability for an event to occur within a given time-bound (after an event) greater or equal than a specified one.

The planned extensions will allow the user to select (from the GUI) a particular BIP engine (the classical engine or the stochastic real-time one) to use. Based on his choice, an adequate property specification language should be used. The tool will still support PBLTL, which is adequate to the classical engine (we plan to enhance the parser and the monitor of this logic to support nested temporal operators). In addition to PBLTL, the tool will offer the possibility to use a real-time temporal logic such as MTL (the choice of the logic to implement is not yet fixed). This implies to implement a new parser for the new logic syntax, and the associated monitoring functionalities over timed BIP traces. Note that the architecture of the tool and its main interfaces will remain unchanged.

5.2.4.6. DESIGN AND TECHNOLOGIES

The BIP-SMC implements several statistical testing algorithms for stochastic systems verification, namely, Single Sampling Plan (SSP), Simple Probability Ratio Test (SPRT), and Probability Estimation (PESTIMATION). Figure 5-13 shows the most important modules of the tool and how they interact together in order to perform statistical model-checking. The tool takes as inputs a stochastic model description in the stochastic BIP format, a PBLTL (Probabilistic Bounded Linear Temporal Logic) property to check, and a set of confidence parameters required by the statistical test.
Figure 5-13. Workflow and architecture of SMC-BIP

During the initial phase, the tool performs a syntactic validation of the PBLTL formula through a parser module. Then, it builds an executable model and a monitor for the property under verification. Next, it will iteratively trigger the stochastic BIP engine to generate execution traces which are monitored to produce local verdicts. This procedure is repeated until a global decision can be taken by the SMC core module (that implements the statistical algorithms). As our approach relies on SMC and since it considers bounded LTL properties, we are guaranteed that the procedure will eventually terminate.

It is worth mentioning that in our implementation, atomic propositions of PBLTL properties are constructed from the system variables. For instance, the PBLTL formula $P_{=7}[G^{1000}(\text{abs}(\text{Master}.tm-\text{Slave}.ts) \leq 160)]$ stands for "What is the probability that the difference between Master variable $tm$ and Slave variable $ts$ is always under the bound 160?". In this example, $\text{Master}.tm$ and $\text{Slave}.ts$ are systems variables pertaining to components Master and Slave respectively. Note that properties specification language offers the possibility to use built-in mathematical functions. In the example above, the $\text{abs}()$ function is used to compute the absolute value of ($\text{Master}.tm - \text{Slave}.ts$).

BIP-SMC is fully developed in the Java programming language. It uses JEP 2.4.1 library (http://www.singularsys.com/jep/index.html, under GPL license) for parsing and evaluating mathematical expressions, and ANTLR 3.2 (http://www.antlr.org/) for PBLTL properties parsing and monitoring. At this stage, BIP-SMC only runs on GNU/Linux operating systems as it relies on the BIP simulation engine (which can also be recompiled to target Mac OSX). The current release of the tool has been enriched with a graphical user interface for more convenience (see Figure 5-14).
The current version also includes supports of the BIP2 language (http://www-verimag.imag.fr/New-BIP-tools.html) while still ensuring compatibility with the previous version. The model checker is available for download from http://www-verimag.imag.fr/Statistical-Model-Checking.html, where additional information (video tutorial) on how to install it and to use it can also be found.

5.2.5. FDIR IMPLEMENTATION AND ANALYSIS

Within ESROCOS we assume that the FDIR behaviour is given by the developer in connection to the desired component lifecycle.

The TASTE interface view defines one or more FDIR functions connected via provided and required interfaces to lifecycle components. These connections cover usually the state of components, some of their attributes values, sending commands, reporting command status, etc.

We envision two possibilities for specifying the behaviour of a FDIR function:

1. The developer provides the functionality for any other TASTE function i.e., using an SDL state machine This possibility allows for the simplest / direct FDIR integration within TASTE models – as everything is done in TASTE.

2. The developer provides the FDIR functionality as a BIP component (atomic or composite), to be plugged (wrapped) within the TASTE function. This possibility is considered because (1) BIP is a priori a more convenient language for expressing FDIR behaviour (complex multi-party interactions, timing constraints, etc.) and (2) FDIR behaviour in BIP will be eventually synthesized automatically from higher level specifications (cf. planned work in OG2 ERGO). Nonetheless, this possibility requires the integration of BIP components into TASTE (see below).

As mentioned above an FDIR component could be generated for a given TASTE design and requirements by the process and tools under development in OG2 ERGO. While ERGO focuses in automatically obtaining such a component when one exists, we are interested in ESROCOS by its validation and verification. We consider the FDIR component given by one of the 2 cases above and we want to check whether the TASTE design including the FDIR component satisfies its requirements under the presence of faults. This analysis is done with BIP via the TASTE2BIP translation and it is detailed below.

5.2.5.1. FDIR ANALYSIS

Regardless the specification of the behaviour, FDIR is subject to validation and verification activities. For analysis scenario, we rely on BIP and associated tools. The analysis will be performed as follows:
• Construct the *nominal model* in BIP: this model corresponds to the TASTE model (including FDIR) and is automatically obtained by translation using TASTE2BIP.

• Construct the *extended model* in BIP: this model extends the nominal model by adding the effect of faults (fault occurrences, and their impact on the behaviour). This construction is semi-automatic – additional models and user expertise is needed.

• *Analyze* the extended model for specific requirements (RAMS, etc.): the analysis are carried out using the validation & verification flows.

Finally, notice that the analysis can be avoided to some extent. That is, whenever the FDIR behaviour has been synthesized such that to fulfill some requirements by construction, there is no need to be re-validated against these requirements.

### 5.2.5.2. FDIR IMPLEMENTATION AND INTEGRATION

FDIR implementation and integration is considered for the situations where the FDIR functionality is not already expressed using a TASTE-compatible language (SDL, C, etc.).

In particular, we shall consider the implementation and integration of FDIR components in BIP within a TASTE design. In this case, the integration shall take place at source code level: the auto-generated code (C++) from the BIP component and the engine libraries (C++) are going to be wrapped as a TASTE component. The approach must correctly “accommodate” the BIP semantics within TASTE in particular, the timely execution of FDIR actions whenever specific conditions are met. These conditions can be checked either periodically (that is, the FDIR component runs on a specific thread) or under specific events (that is, the FDIR component is notified by lifecycle components about specific events).

#### Figure 5-15. Use Case diagram for FDIR implementation and analysis

### 5.2.5.3. USE CASE “IMPLEMENT FDIR IN TASTE”

Description: This use case covers the modelling of FDIR functionality in TASTE.

Actor: The developer.
Input: A TASTE interface view containing an FDIR function connected to the concerned lifecycles.
Output: The TASTE interface view completed with an SDL state machine for the FDIR function.

Basic flow: Based on the connected lifecycles, the developer models an SDL state machine describing the FDIR behaviour in case of erroneous behaviour, e.g., restarting or stopping a component.

5.2.5.4. USE CASE “RUN V&V”
Description: A TASTE model with an FDIR function can be subject to verification and validation. The V&V can be performed with the BIP tools via the TASTE2BIP translation.
Actor: The developer.
Input: A TASTE model including FDIR functionality.
Output: Possible executions, SMC validation or invariant verification with iFinder of safety properties.
Sub-cases: This use case is the generic description of the 3 possible V&V activities the BIP tools offer. TASTE models via BIP can be subject to: (1) running simulations (use case “Run simulation” from the BIP Compiler and Engine module), (2) SMC analysis for the specified requirements (use case “Run SMC” from the SMC-BIP module), and (3) safety properties verification with iFinder (use case “Run verification” from the iFinder module reconsidered and extended in OG2).

5.2.5.5. USE CASE “IMPLEMENT FAULTS BEHAVIOUR IN BIP”
Description: In order to perform V&V on the conceived TASTE model, the faults model hardware components can exhibit is required. The TASTE and faults models are taken together and with a (probabilistic) fault injection mechanism are evaluated. This use case covers the modelling of faults behaviour.
Actor: The developer.
Input: The list of hardware equipment which might exhibit faults.
Output: A BIP model of the faults behaviour for each item of the list.
Basic flow: The developer inspects the list and the manual of each item (provided by the manufacturer) and designs an individual faults behaviour.

5.2.5.6. USE CASE “IMPLEMENT AND INTEGRATE BIP FDIR IN TASTE DESIGN”
Description: A second option for FDIR modelling is using BIP and encapsulating the FDIR component and BIP Engine in a TASTE FDIR component. This use case covers the BIP modelling of a TASTE FDIR component.
Actor: The developer.
Input: An open BIP FDIR model.
Output: A TASTE design containing the given BIP model and engine inside of an FDIR component.
Remarks: This component could be the one synthesized in OG2 with the FDIR approach. The complexity of this use case comes from the encapsulation of the BIP FDIR component and engine in a TASTE component, as explained above.
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 and utility functions. The TASTE infrastructure then generates the necessary serialization functions for data transport.

The base types extend the TASTE basic and extended types, which define the basic numeric, Boolean and string types. The following paragraphs explain the logic of the base-types using UML. As ASN.1 is not object oriented, the inheritance relationships are mapped by “flattening” the corresponding types into flat ASN.1 structures. The dependency relationships are mapped to ASN.1 by including a field of one data type inside the other. Finally, variable-sized types in C++ are mapped to fixed- or bounded-size types in ASN.1.
The common data types are organized in several groups. The first group is the timestamps and sensor data types. Two classes from this group have special relevance:

**Time**: since robotics is all about managing physical systems that evolve in a physical environment, and modify it, the relationship between the data that are being processed and this environment is central to the problem of data processing. Key to that is the ability to associate data temporally: the ability to mark when a particular information was sensed from the physical world. This process is usually referred as “data timestamping”. Time class codifies as single 64-bit integer representing microseconds from 1970-01-01T00:00:00.

**Joint-state**: robot are electro-mechanical devices. The basic structure of such mechanical complexity is the Joint. The JointState class encodes the primary information in order to sense and command information from/to a Joint driver.

The next group is the algebraic data types shown in Figure 5-17 and Figure 5-18. This group describes the data types to send spatial information in a 3D world. The group describes the Special Orthogonal Group SO(3) and the Special Euclidean Group SE(3) as the basic data representation to provide translation and rotation on manifolds. Figure 5-18 depicts the part 2 of this group of data types which among other features they include uncertainty information in SO(3).
The most relevant class in the algebraic group is `Transform3d`. This class describes a translation and rotation on Affine spaces. It allows rotation, translation, reflection and scaling in a homogeneous matrix (rotation matrix and a translation vector).

**Orientation:** this class describes a rotation in SO(3). Instead of using rotation matrices as in `Transform3d` the orientation uses a quaternion encoded in a 4D vector of Double. Quaternion is a convenient representation of orientations and rotations of objects in three dimensions. It is compact, efficient and stable against singularities. The first element of the vector stores the scalar part of the rotation while the vector part is in the last 3x1 elements.

**Figure 5-17. Algebraic base types to describe SO(3) and SE(3) – part 1**
Figure 5-18. Algebraic base types to describe SO(3) and SE(3) - part 2

The commands data type are described in Figure 5-19. Those are timestamped data types which are commonly used in output ports. The most relevant type in mobile robotics is **Motion2D**. This class gives the basic information to a ground robot regarding the next pose (position and rotation) to achieve.

Figure 5-19. Command base types to send control information to robots

The samples group entail the types related to sensory data. They provide the necessary information regarding robot internal state. The most commonly used type is the **RigidBodyState**. This type described timestamped information regarding a rigid body pose and velocity together with the uncertainty.

**IMUSensors**: this sequence groups inertial data information typically given by an Inertial Measurement Unit (IMU). An IMU is a very common sensor in robotics, providing timestamped 3D information of angular velocity, acceleration, and magnetic field when available.

**Joints**: The joints state is a NamedVector if JointStates accessible by a unique identifier (e.g. string). Therefore, the class provides the same information as JointState but group in a sequence of elements. Joints typically reveals relevant proprioceptive information of the robot.

Figure 5-20 below depicts the class diagram concerning the samples data type group.
Figure 5-20. Base types for data samples

**Pseudo-dynamic sized data structure:** In TASTE all types which are used must be defined as fixed sized types. Dynamic sized types such as vectors with no pre-initialized length are not supported. This constraint complicates the use of basic robotic types shared between components and also systems. Imagine a two robot systems A and B, where both robots have a manipulator, but the manipulator of A has 6 and the manipulator of B 7 degrees of freedom. Still we want to be able that the both systems might use the same joint interpolation component – implementing a 6 DOF version and separately a 7 DOF version appears to be duplicate work. In order to allow scenarios like the one described, a framework mechanism must be provided that allows "Pseudo-dynamic sized data structures". Component Developers must be allowed to develop components that used
pseudo-dynamic-sized types without having to know the size of the data structures in the final application. But still, it must be possible to inject the missing information later on in the system integration phase without rewriting of any source code.

We are evaluating different possible approaches to this problem. A first approach is to establish support for ASN.1’s "Parametrized Types" feature within TASTE, as illustrated in Figure 5-21. Currently such types are not foreseen within the TASTE workflow and currently underspecified types (such as Joints in the Figure) would be lost in the translation to AADL.

Another option that we are evaluating is facilitating special template types, that specify there under-specified attributes (cf. size in Figure 5-22). The template types are, from an ASN.1 perspective, fully specified. By convention, it is expected to inject the missing information at system integration time with AADL context parameters. A specific preprocessor tool analysis the resulting AADL file for Template types and corresponding context parameters and generates fully specified fixed length container types as replacements for the template types.

The final method of handling pseudo-dynamic types is no yet decided, but will be later on within the project.

The Table 5-1 summarizes the base types provided by ESROCOS and used for interfacing from the applications with the different components of the framework.
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Description</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Time stamp codified as single int64_t number representing microseconds from 1970-01-01T00:00:00</td>
<td>Time stamps information</td>
</tr>
<tr>
<td>VectorXd</td>
<td>Static size vector of 2/3/4/6 dimensions of double type</td>
<td>Dynamics/ Kinematics/ SLAM</td>
</tr>
<tr>
<td>MatrixXd</td>
<td>Static size squared matrices of 2/3/4/6 dimension of double type</td>
<td>Dynamics/ Kinematics/ SLAM</td>
</tr>
<tr>
<td>VectorXf</td>
<td>Static size vector of 2/3/4/6 dimensions of float type</td>
<td>Dynamics/ Kinematics/ SLAM</td>
</tr>
<tr>
<td>MatrixXf</td>
<td>Static size squared matrices of 2/3/4/6 dimension of float type</td>
<td>Dynamics/ Kinematics/ SLAM</td>
</tr>
<tr>
<td>Point</td>
<td>Vector3d point</td>
<td>Arbitrary Way-Points</td>
</tr>
<tr>
<td>Angle</td>
<td>Rotational angle in radians</td>
<td>Single angular sensor or joints</td>
</tr>
<tr>
<td>Quaterniond</td>
<td>Static size Vector4d of the form w+xi+yj+zk.</td>
<td>Rotations in SO(3) group</td>
</tr>
<tr>
<td>Transform3D</td>
<td>Static size Matrix4d of the form [R</td>
<td>t]</td>
</tr>
<tr>
<td>Pressure</td>
<td>Pressure in Pascals</td>
<td>Pressure sensors</td>
</tr>
<tr>
<td>Temperature</td>
<td>Static size temperature stored in Kelvin (IS unit)</td>
<td>Sensor temperature</td>
</tr>
<tr>
<td>JointState</td>
<td>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]</td>
<td>Motor temperature</td>
</tr>
<tr>
<td>Twist</td>
<td>Vector3d linear velocity [m/s] and Vector3d angular velocity [rad/s]</td>
<td>Inverse and direct kinematics</td>
</tr>
<tr>
<td>Wrench</td>
<td>Vector3d force [Newtons] and Vector3d torque [Newton meter]</td>
<td>Dynamic control</td>
</tr>
<tr>
<td>Pose</td>
<td>Position(Point [meters]) and orientation(Quaterniond) of a robot pose.</td>
<td>Localization and Planning</td>
</tr>
<tr>
<td>AUVMotion</td>
<td>Motion command for Underwater vehicle</td>
<td>Dynamic control</td>
</tr>
<tr>
<td>BodyState</td>
<td>State of a rigid body state [Pose] in SE(3) with uncertainty information</td>
<td>State estimation/SLAM</td>
</tr>
<tr>
<td>DepthMap</td>
<td>DepthMap image from sensory data (e.g. LIDARs)</td>
<td>Perception/SLAM</td>
</tr>
<tr>
<td>DistanceImage</td>
<td>2D array structure representing a distance image for a pinhole camera model [pixels].</td>
<td>Perception/SLAM</td>
</tr>
<tr>
<td>Frame</td>
<td>Data structure representing a visible camera image and its metadata [pixels].</td>
<td>Perception/SLAM</td>
</tr>
<tr>
<td>IMUSensors</td>
<td>Information from an Inertial Measurement Unit(IMU) acceleration in [m/s], gyroscopes [rad/s] and magnetometers [Tesla]</td>
<td>Control, Localization and Mapping</td>
</tr>
<tr>
<td>Data Type</td>
<td>Data Description</td>
<td>Application</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>JointLimitRange</td>
<td>Specification of Joint limits for a robotic model position [double], speed[position/s], effort [N or Nm], raw (i.e. PWD signal) and acceleration[rad/s^2 or m/s^2]</td>
<td>Control, simulation</td>
</tr>
<tr>
<td>JointLimits</td>
<td>Static size vectors of JointLimitRange</td>
<td>Control</td>
</tr>
<tr>
<td>Joints</td>
<td>Static size vector of JointState for multiple optionally named joints.</td>
<td>Control and state estimation</td>
</tr>
<tr>
<td>JointsTrajectory</td>
<td>Structure to hold a time-series in the form of vector of JointState, for multiple optionally named joints.</td>
<td>Control, planning</td>
</tr>
<tr>
<td>JointTransform</td>
<td>Defines the frame transformation provided by a Joint. Helper data type between Joint and Transform3D</td>
<td>Control, state estimation</td>
</tr>
<tr>
<td>LaserScan</td>
<td>Laser scans measurements from a laser sensor</td>
<td>Perception</td>
</tr>
<tr>
<td>Motion2D</td>
<td>Motion command for ground vehicles moving in a 2.5D space</td>
<td>Control and planning</td>
</tr>
<tr>
<td>Pointcloud</td>
<td>Static size vector of Points</td>
<td>Perception and Mapping</td>
</tr>
<tr>
<td>RigidBodyAcceleration</td>
<td>RigidBodyState with acceleration information</td>
<td>State estimation</td>
</tr>
<tr>
<td>RigidBodyState</td>
<td>State of a rigid body state [Pose] in Affine3d with uncertainty information per each element of the Pose (position and orientation)</td>
<td>State estimation</td>
</tr>
<tr>
<td>Sonar</td>
<td>Representation of data acquired by a Sonar sensor</td>
<td>Perception</td>
</tr>
<tr>
<td>TimeMark</td>
<td>Time stamp helper structure to compute, the time that has passed since the recorded time and now.</td>
<td>Time stamps information</td>
</tr>
<tr>
<td>Trajectory</td>
<td>Trajectory defined in a Spline form.</td>
<td>Planning and control</td>
</tr>
<tr>
<td>TransformWithCovariance</td>
<td>Transform3D with uncertainty</td>
<td>Affine 3D Transformations</td>
</tr>
<tr>
<td>TwistWithCovariance</td>
<td>Twist with uncertainty information</td>
<td>Inverse and direct kinematics, State estimation</td>
</tr>
<tr>
<td>Waypoint</td>
<td>Points representation for a Pose in a path</td>
<td>Planning</td>
</tr>
</tbody>
</table>

A set of utility functions will be 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.
In TASTE, data types are defined within the system workspace as ASN.1 structs, which are converted to AADL files by the `asn2aadl` tool. Figure 5-23 shows the directory tree of an ESROCOS workspace to illustrate the inclusion of types from the different components.

### 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.20] for more information: [http://opencv.org/](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.21] for more information: [http://eigen.tuxfamily.org/](http://eigen.tuxfamily.org/)

### 5.3.4. TRANSFORMER LIBRARY

The transformer library provides the functionality of representing a graph with coordinate frames as nodes and static frame transformation operations or dynamic transformation generators as edges. Given this graph and valid data from the dynamic transformation generators, transformations between arbitrary frames can be queried. The path along the edges that need to be traversed between the frames correspond with the arithmetic frame transformation operations that must be applied in order to get the requested result.

The transformer library will be implemented similar to its original version coming from rock (see also [https://github.com/rock-core/drivers-transformer](https://github.com/rock-core/drivers-transformer)). The library will provide a C interface and a memory allocation-free operation mode. The memory allocation free operation mode will requires the specification of the transformation graph before runtime, as static configuration structures such as context parameters.
5.3.5. STREAM ALIGNER LIBRARY

Another common problem in the robotic domain is to handle data that might be created and processed asynchronously. To support the handling of such data, ESROCOS will implement a mechanism to buffer multiple asynchronous data streams for the selection of best corresponding samples at a given time point, inspired by the *stream aligner* [RD.22] in ROCK.

The design of this library will be based in the equivalent from the ROCK ecosystem. The library will be developed according to the space quality level requirements.

5.3.6. PUS SERVICES

5.3.6.1. OVERVIEW

The PUS Services component (PUS_SERVICES) is responsible of the handling of PUS Services messages at Space Segment.

PUS_SERVICES should be understood as a middleware between the Space Segment Architecture and the well-known PUS Services standard (ECSS-E-ST-70-41C, April 2016) [RD.17], so a brief description of the major and most significant aspects of this well-known standard are below provided.

This Standard introduces the concept of PUS services, consisting of PUS subservices. The services and subservices formalise the closely related and self-contained set of space system functions and all related entities and interaction artefacts.

Each PUS subservice is composed of PUS subservice entities, each one playing either the role of a subservice provider or the role of a subservice user. Each PUS subservice entity is hosted by an application process on-board or on-ground.

![Figure 5-24. The space to ground PUS service system context](image)

As depicted in Figure 5-24, it is usually understood that the on-board application processes host the subservice providers and the ground application processes the subservice users but this standard does not constrain those relationships.
The information exchanged between a subservice user and subservice provider is termed a "message". A message is transmitted semantically unchanged by the transmission protocol that connects the subservice users and subservice providers.

A message sent by a subservice user to a subservice provider, to invoke the execution of on-board activities, is termed a "request". Each request contains one or more instructions, one for each activity to execute.

A message sent by a subservice provider to a subservice user is termed a "report". Each report contains one or more notifications.

5.3.6.2. SERVICES IDENTIFICATION

Although at PDR stage our preliminary design is presented using UML diagrams, PUS Services are to be provided in the form of TASTE functions for CDR. Their interface will comply with the ECSS Standards at interface level (Not protocol level). The following are the PUS Services and Subservices will be implemented and provided as a C/C++ library (PUS-R14). Additional Subservices will be implemented if time allows.

<table>
<thead>
<tr>
<th>Sub-service</th>
<th>Service requests</th>
<th>Sub-service</th>
<th>Service reports</th>
<th>To be provided</th>
<th>Req. Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Verification service -1</td>
<td></td>
<td></td>
<td>1 successful acceptance verification report</td>
<td>Y</td>
<td>PUS-R09</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 failed acceptance verification report</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device Access service -2</td>
<td></td>
<td></td>
<td></td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Housekeeping service -3</td>
<td></td>
<td></td>
<td>2 create a diagnostic parameter report structure</td>
<td>Y</td>
<td>PUS-R02</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 delete diagnostic parameter report structures</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7 enable the periodic generation of diagnostic parameter reports</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8 disable the periodic generation of diagnostic parameter reports</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25 housekeeping parameter report</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>26 diagnostic parameter report</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter Statistics Reporting service -4</td>
<td></td>
<td></td>
<td></td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Event Reporting service -5</td>
<td></td>
<td></td>
<td>1 informative event report</td>
<td>Y</td>
<td>PUS-R03</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 low severity anomaly report</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 medium severity anomaly report</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 high severity anomaly report</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory Management service -6</td>
<td></td>
<td></td>
<td></td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Function Management service -8</td>
<td></td>
<td></td>
<td>1 perform a function</td>
<td>Y</td>
<td>PUS-R04</td>
</tr>
<tr>
<td>Time Management service -9</td>
<td></td>
<td></td>
<td>1 set the time report generation rate</td>
<td>Y</td>
<td>PUS-R05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 CUC time report</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time-Based Scheduling service -11</td>
<td></td>
<td></td>
<td>1 enable the time-based schedule execution function</td>
<td>Y</td>
<td>PUS-R08</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 disable the time-based schedule execution function</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Sub-service | Service requests | Sub-service | Service reports | To be provided | Req. Trace
--- | --- | --- | --- | --- | ---
3 | reset the time-based schedule | | | | |
4 | insert activities into the time-based schedule | | | | |

**On-Board Monitoring service -12**

1. enable parameter monitoring definitions
2. disable parameter monitoring definitions
12. check transition report

**Large Packet Transfer service -13**

**Real-Time Forwarding Control service -14**

**On-Board Storage and Retrieval service -15**

**Test service -17**

1. perform an are-you-alive connection test
2. are-you-alive connection test report

**On-Board Control Procedure (OBCP) service -18**

1. direct-load an OBCP
2. unload an OBCP
3. activate an OBCP
4. stop an OBCP
5. suspend an OBCP
6. resume an OBCP
12. abort an OBCP

**Event-Action service -19**

1. add event-action definitions
4. enable event-action definitions
5. disable event-action definitions

**Parameter management (additional service 20)**

1. report parameter values
2. parameter value report
3. set parameter values

**Request Sequencing service -21**

**Position-Based Scheduling service -22**

**File Handling service -23**

2. delete a file
14. copy a file
23. file copy status report

---

**5.3.6.3. PUS_SERVICES TECHNICAL OVERVIEW**

PUS_SERVICES covers and supports the way Space system functions are defined when involved in interactions between space and space components and/or space and ground components.
Above figure describes the following major drivers taken into account for its design:

- **PUS_SERVICES is provided as a library** belonging to the “Core Space Robotic Functions” (see Figure 4-1).

- **PUS_SERVICES will act as a PUS Service-Subservice provider** for other Space Application components or Ground Segments (GS) which act as PUS Service-Subservice users.

- **PUS_SERVICES provides four differentiated interfaces (3 external and 1 internal):**
  - **Event_IF** (external): intended to allow interactions from other Space Application components related to Events operations (including Event-Reporting and Event-Actions functions).
  - **TC_IF** (external): intended to allow the interactions from other Space Application components and/or Ground Segment. This interface will provide functions to all messages-requests (TCs) defined in the PUS_API, excluding those related to Events handling, which are provided in the Event_IF interface. This interface also provides functions for the OBCP handling TC(18,X).
  - **TM_IF** (external): intended to allow the interactions from other Space Application components and/or Ground Segment. This interface will provide functions to all messages-reports (TMs) defined in PUS_API, excluding those related to Events handling, which are provided in the Event_IF interface.
  - **OBCPE_IF** (internal): intended to allow internal interactions from PUS_API component related to the handling of the OBCP execution environment.
PUS_Service preliminary design has been split in four major sub-components:

- **PUS_API**: it provides an API for the set of PUS services operations identified as needed for the ESROCOS framework.
- **Event_Handler**: it is intended to support the handling of Events-reports and the handling and triggering of Event-Actions features required by the ESROCOS framework, making transparent to other Application components the PUS services operations required for its implementation.
- **OBCP_Engine**: it is intended to support the execution of the OBCPs in the ESROCOS framework. It includes the OBCP execution environment defined in more detail in section 5.3.6.6.
- **DataPool_API**: it must be understood as a library or set of libraries responsible of the handling and manipulation of the different containers.

### 5.3.6.4. PUS_API

This component is intended to provide an API (Application Programming Interface) to the final PUS(Services, Subservices) implementation in ESROCOS. It serves as an interface for the associated PUS TC&TM. It contains the basic functions to be called once a TC is received, as well as the main functions to be called in order to send TM.

![PUS_API preliminary design](image)

**Figure 5-26. PUS_API preliminary design**

### 5.3.6.5. EVENT_HANDLER

This component is intended to allow interactions from other Space Application components related to Events operations (including Event-Reporting and Event-Actions functions).
Furthermore, the above figure describes how Event_Handler delegates the implementation of its provides function to PUS_API component (PUS(05,XX) and PUS(19,XX)) and how it also makes usage of the DataPool_API library to get/set data from/to Event_Actions container.

5.3.6.6. OBCP_ENGINE

An OBCP (On-Board Control Procedure) represents a procedure executed on-board the spacecraft that can be easily loaded, executed or replaced on-board without modifying the remainder of the on-board software [RD.27].

There are two types of OBCPs:

- **OBAPs** (On-Board Application Procedures) are procedures which form part of the on-board application software. Simply, some basic functions of the application software are implemented by means of OBCP. The overall qualification of the spacecraft requires the integration of the complete set of OBAPs.

- **OBOPs** (On-Board Operations Procedures) are designed for operational purposes to allow flexible change and upload during mission. They are not involved in the qualification of the spacecraft.

The OBCP system consists of a preparation environment located on ground and an execution environment which includes a ground element and an on-board one:

- The preparation environment lets the operations engineers write and verify the procedures.

- The execution environment located on-ground serves to command and monitor the execution of the OBCPs.

- The execution environment located on-board includes the execution engine and the supporting services.

The next figure shows a preliminary design of the OBCP_Engine at PDR stage.
On-Board control procedures (OBCPs) are software operations which are interpreted at run-time by an OBCP_Engine. The engine typically provides the capability to load, delete, start, stop and abort OBCPs delegating this to the **OBCP_Manager**, **OBCP_Scheduler** and **OBCP_Interpreter**.

The figure also describes the following scenarios:

- Instances of the OBCP_Engine are handled through the OBCPE_IF interface, which is accessed from PUS_API::TC(18,X).
- OBCPs could be dependent of Events produced in the system, so each time an Event is handled by the Event_Handler, it will notify to the OBCP_Engine about it.
- At the same time, an OBCP execution could generate Events, so this must raise it on the Event_Handler.
- OBCP execution also needs interacts with the PUS_API component in order to request TC and TM.

### 5.3.6.7. OBCP PROGRAMMING LANGUAGE

Spacecraft and Space Robots must run on HW platforms with reduced resources as RAM and CPU and on SW consisting of low-level drivers, a middleware platform, and high-
level application software. The low-level and middleware parts of this stack are written in C and/or Ada, are well proven and relatively stable across missions.

On the other hand, the high-level application software has a tendency to require regular updating while the spacecraft is in flight. Such flexibility is hard to meet using C and/or Ada which are statically linked and require recompiling the entire flight stack and patching the existing live software. Patching can be a dangerous operation and it is desired to reduce or eliminate the need for it.

Recently, it is desired to write the high-level application software in a language more suited to the task, in other words a high-level language.

We propose to use **MicroPython** taking advantage of the work already done in the ESA study *"Porting of MicroPython to LEON platforms"* [RD.28]. This approach offer us the following advantages:

- MicroPython is a recent and independent implementation of the Python language which is written in C and optimized to run on constrained systems, such as microcontrollers and embedded devices.

- Python has extensive support for exceptions and exception handling, and this makes it easier to write very robust programs. There is also a clean separation between different data types (for example there are distinct float and integer types) and operations on these types are simple and well defined, leading to less errors when manipulating variables. There are no "primitive" errors in Python because there are no pointers, integers are arbitrary precision, and strings and arrays always have their bounds checked.

- The Python language is typically implemented by compiling scripts to bytecode and then executing the bytecode within a Python virtual machine (VM). The VM is supported by the Python runtime which provides functions for operations like hash-table lookup and string manipulation.

- In the development of MicroPython there are three main principles that are used: 1) keep memory (RAM) usage to an absolute minimum; 2) keep code size (ROM) to a minimum; 3) make it as efficient as possible. The principles make MicroPython well suited to bring Python into space, and it already targets machines that have fewer resources than a typical spacecraft. The problem of determinism is also made less because MicroPython uses only the stack for most of the basic Python language features, thus reducing or eliminating the need for a dynamic heap.

- MicroPython has many configuration options which are set and tuned at compile time and allow it to fit and run in a large variety of systems. Each such version of MicroPython is called a "port". One of these ports is the Unix/Window port which runs on a normal desktop PC (and is used for testing purposes, among other things).

- The goal of the mentioned study was based in create an implementation of MicroPython that runs under the RTEMS operating system on LEON hardware. Special consideration was given to the following points:
  - to optimize the use of both RAM and CPU, to have low use of resources;
  - to provide determinism with respect to resource allocation, in particular deterministic memory allocation and/or bounded dynamic memory allocation and de-allocation;
  - to provide an interface between Python and the C and Ada languages, so that Python can call C/Ada and vice-versa;
  - to have the ability to execute Python scripts in parallel;
- to create a prototype of an OBCP execution environment which utilizes the MicroPython VM to execute control scripts.

So, after all aforementioned, it looks like that MicroPython would be a good solution for the OBCP programming language, because it is already tested and checked for the ESROCOS target environment and it could be used for creating self-contained Python tasks which are low on resource usage and implement an On-Board Control Procedure (OBCP). These Python OBCPs would typically be small scripts that execute routine operations and interact in a safe and controlled manner with the rest of the system.

5.3.6.8. DATAPOOL_API

This component is intended to provide an API (Application Programming Interface) to a set of containers identified as needed taking into account ESROCOS needs.

Figure 5-29. DataPool preliminary design

This component will be initially accessed by the sub-components of the PUS_SERVICES components (Event_Handler, PUS_API and OBCP_Engine), but it could be externalised to allow its usage from another Application components.

At the same time, the above figure shows how class DataPool_Handler plays the role of a unique front-end for all functions accessing and modifying the container items, but it could be split in specialised front-ends (one for each container) in order to minimise the final binaries sizes. In any case, DataPool_Handler must guarantee the data consistency in parallel accesses.

Finally, it should be noted that containers could be finally implemented as FIFO queues, sets, maps or another container satisfying the requirements of the ESROCOS framework.
5.4. DEPLOYMENT AND EXECUTION OF APPLICATIONS

5.4.1. AIR HYPERVISOR

The overall architecture of AIR hypervisor will be affected at two fronts

1. The AIR tools that will parse the produced outputs of TASTE and from there execute the required processing to generate a TSP binary to be deployed on a target

2. The inclusion as a POS (Partition Operating System), the same RTOS being used in TASTE, namely RTEMS.

5.4.1.1. AIR TOOL CHAIN ADAPTATION TO TASTE

The partition management kernel of AIR is fairly simple, in fact most of the intelligence and complexity of AIR resides in its tool chain.

![AIR Tool Chain Diagram](image)

**Figure 5-30. AIR Tool Chain**

The AIR tool chain has the objective of:

- Hiding the complexity to the user
- Generate and rearrange the source code for a TSP environment
- Automate the makefile system
The AIR tool chain will be the interface in TASTE, TASTE will produce source code (pointed with blue arrow in Figure 5-30) and the ARINC 653 compliant TSP configuration (pointer with an yellow arrow in Figure 5-30).

Therefore is now required to detect the changes on both elements in comparison to what the tool-chain is now expecting.

The source code is expected to be different because TASTE structures those files according to the interface view component structure. The tool chain will then be upgraded to be able to read the source code in this structure, transform to the structure that is currently used by AIR and adapt the automation of makefiles accordingly.

The AIR tool chain currently structures the source code in a set of folders, where each folder contains all the information concerning each partition.

In principle, the produced configuration of TSP should not have major differences since both AIR and TASTE are obliged to follow the ARINC 653 specification. Nevertheless, it is normal for this configuration to include additional information that is specific to TASTE or the hypervisor. The tool chain will then be adapted to recognize those additional features and bring them to good use.

5.4.1.2. UPGRADE OF RTEMS

Any RTOS to be used as a Partition Operating Systems, requires some adaptations in order to integrated into the hypervisor. These adaptations consists in:

- Disabling of several functionalities of the RTOS that are replaced by similar functionalities that now are controlled by the AIR, namely trap table, memory access and time related functionality.
- Ensure the critical errors detected by RTOS are now handled by the hypervisor bringing the bridge to AIR Health Monitor
- Code changes to ensure correct build of the RTOS within the hypervisor due interference of redefinition of same elements on both RTOS and AIR

Unfortunately such adaptation differs from RTOS to and from version to version of same RTOS.

In the specific case of ESROCOS, the EtherCAT and CAN device drivers of RTEMS will be qualified, in consequence those devices drivers must be used also by the AIR hypervisor.

The handling of the device drivers in AIR is done through the solution of sharing I/O from a server named I/O partition which allows extensibility for support of multiple devices and field buses.

Technically this means that the implementation and management of device drivers is occurs in a single place of the hypervisor where RTEMS related functionality is migrated to AIR I/O partition.

AIR will then require to upgrade its RTEMS to use the exact same device drivers that will be qualified, in order to achieve this requirement there are several possible approaches:

1. Upgrade the current version of RTEMS 4.8 impr to version RTEMS 4.10 of TASTE
2. Replace device drivers of RTEMS 4.8 impr with the qualified device drivers achieving a fully qualified RTOS
3. Maintain the current RTEMS 4.8 impr, and qualify those drivers and support them on TASTE

The intended approach is solution 1, but it is challenging and its conclusion is dependable of on the amount changes done on RTEMS since 4.8 impr. Those change are quite
considerable, first because \textit{RTEMS 4.8 impr} is a stripped down to the minimum of the RTEMS 4.8 version and second from RTEMS 4.8 to 4.10 considerable changes have been done.

Those changes are illustrated bellow with start.S file (the SPARC processor startup), the file is compared between RTEMS 4.8 (left) in RTEMS 4.8 impr (right), in red is the code that has been stripped down. The image only shows the first lines of the file start.S, the bars on the left show in colours what has been changed in the entire file.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{comparison.png}
\caption{Comparison of RTEMS 4.8impr vs RTEMS 4.8 (start.S file)}
\end{figure}

The following pictures illustrates how further challenging it is the upgrade to RTEMS 4.10, by comparing the same start.S file in version 4.8 against 4.10.
Figure 5-32. Comparison of RTEMS 4.8impr vs RTEMS 4.8 (start.S file)

Hence it has also been considered the other approaches since the total upgrade to RTEMS 4.10 may not be feasible within the schedule of the project.

As initial tasks, the major files such start.S and bootcard files will be upgraded, and based on the identified difficulties, impact on the hypervisor and time taken, a decision will be taken on the final approach to take.

5.4.2. HAIR EMULATOR

Similarly to the AIR hypervisor, the same fronts must be considered namely the HAIR toolchain and RTEMS upgrade.

Regarding the RTEMS upgrade there is no issue on this particular case, at laboratory grade level it will not be used a SPARC RTEMS version but instead POSIX RTEMS. Also the qualified drivers will not be used, being replaced by the device drivers offered by the underlying Linux operating system. Therefore the POS will be maintained as it is.

Concerning the HAIR toolchain, it is 99% equal to AIR toolchain, and therefore it will undergo the same activities described in 5.4.1.1.
5.4.3. CAN BUS DRIVER

In order for TASTE to 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).

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 5-33. Class diagram for the PolyORB-HI CAN driver

Figure 5-34. Typical interface view used for CAN driver
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, please refer to section 5.4.1.2.

**5.4.4. ETHERNET/ETHERCAT DRIVER**

Although the PolyORB-HI-C distribution already includes an Ethernet driver, a new version is developed for ESROCOS using the new RTEMS DriverManager and bsdnet libraries. On top of this driver, a set of capabilities will be added in order to support the functionality of an EtherCAT master node.
The PolyORB-HI Ethernet driver is represented in Figure 5-36 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 5-37. Typical TASTE Interface View used for Ethernet driver

Figure 5-38. 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.
5.4.5. 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.

The PolyORB-HI SpaceWire driver is represented in Figure 5-39 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

![Figure 5-39. Class diagram for the PolyORB-HI low-level SpaceWire driver](image)
- **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 5-40. Typical TASTE Interface View used for SpaceWire driver**

**Figure 5-41. TASTE Deployment View specifying SpaceWire driver and bus**
5.5. MONITORING, DEBUGGING AND TESTING

5.5.1. DATA LOGGER

The Design of the data logging function is not yet clarified. Possible approaches are for instance:

- Explicit connections to all Provided Interfaces within an application in Interface View
- Automatic hidden connections to all Provided Interfaces
- Compilation of logging functionality in the deployment binaries
- Low-level logging on middleware level
- External logging piggy-backing on the RCOS bridges

All of these approaches will be discussed and evaluated later on in the project.

5.5.2. VIZKIT3D INTEGRATION

vizkit3d is a GUI application based on the Qt toolkit and Open Scene Graph (OSG). It provides a series of plugins that can render different types of robotics data in a 3D view. Each plugin receives data in the formats defined by the ROCK base-types library. Plugins also provide some configuration properties that determine how the data is rendered.

For integration in TASTE, each plugin is enclosed in a TASTE function. The function has one or more sporadic Provided Interfaces for receiving the data updates to be rendered.

The configuration properties of the vizkit3d window and the plugins is set by means of a configuration file in YAML format. As plugins may have many complex properties, using a configuration file has been deemed preferable to exposing the plugin properties through TASTE context parameters. Instead, each plugin function in TASTE will have two context parameters that define the configuration file and the identifier of the plugin instance.

An example of this is shown in Figure 5-42. It presents a simple model in which a Vizkit plugin function named vizkit_RigidBodyState (corresponding to a RigidBodyStateVisualizationPlugin) renders the data produced by a periodic function called rbsProducer.
The RigidBodyStateVisualization plugin renders a 3D model in the 3D space according to the state vector (position, orientation, etc.) provided by an external source. In the example, the represented object is an IMU device.

The executable can be built from the model using the TASTE infrastructure. When run, the executable opens a vizkit3d window as shown in Figure 5-43, and the image is updated with the data generated by the rbsProducer.

Figure 5-42. vizkit3d plugin in TASTE IV (detail of sporadic PI)

Figure 5-43. vizkit3d window running in TASTE application
The configuration of the vizkit3d window is defined by the configuration file, in YAML format, shown in Listing 5-1. In this example, two plugins are selected: the above mentioned RigidBodyStateVisualization plugin, plus a GridVisualization plugin that draws a grid in the ground plane. This grid plugin does not receive data updates at runtime, so it does not have a corresponding TASTE function block.

**Listing 5-1. Sample vizkit-taste configuration in YAML**

```yaml
# Window (title, geometry)
window:
  title: taste-vizkit3d - body state visualization test
  x: 100
  y: 150

# 3D widget
widget:
  manipulator: TRACKBALL_MANIPULATOR

# Plugins
plugins:
- type: BodyStateVisualization
  name: BS
  modelPath: stim300.stl
  size: 2.0
  frame:
    base: world_osg
    position: [1, 1, 1]
    rotation: [1, 0, 0, 0]
    # rotation quaternion in order [w, x, y, z]

- type: RigidBodyStateVisualization
  name: RBS
  modelPath: stim300.stl
  frame:
    base: world_osg
    position: [1, 1, 1]
    rotation: [1, 0, 0, 0]
    # rotation quaternion in order [w, x, y, z]

- type: ModelVisualization
  modelPath: stim300.stl

- type: GridVisualization
```

The configuration file contains a section for the window characteristics, a section for the 3D rendering widget (which defines the view camera), and a plugins section with a list of plugin instances with their type, identifier and properties. Each plugin function in the TASTE has an identifier, declared as context parameter, that maps to the appropriate plugin instance in the configuration file.

The vizkit3d integration in TASTE consists of the following components:

- **taste-vizkit3d**: a set of TASTE functions corresponding to each vizkit3d plugin that contains a data update function.
- **vizkit3d_c**: a C/C++ library that manages the lifecycle and data input of the vizkit3d application and provides a C wrapper interface for integration with TASTE.
- **asn1-types**: the ASN.1 robotics data types library.
- **asn1-types-support**: a support library for conversions between the C++ types used by vizkit3d and the C types used by TASTE.

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

- **vizkit3d**: the vizkit3d visualization application.
- **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 5-44 presents the architecture and the dependencies among these components.

**Figure 5-44. Component architecture of the vizkit3d-TASTE integration**

The vizkit3d application provides thirteen basic plugins. These are divided in three 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 5-45. 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 5-45. vizkit3d plugin functions in TASTE](image)

In the frame of the ESROCOS project, the integration of vizkit3d components in TASTE models, initially developed in the frame of the SARGON project, will be adapted to:

- Support the newer version of TASTE
- Take advantage of the ROCK middleware bridge, so that a TASTE application can visualize data using an external vizkit3d instance running on RTT
- Incorporate any new vizkit3d features or additional plugins

5.5.3. INTEGRATION OF ROS ASSETS

The main assets from the ROS ecosystem that have been identified for integration in the ESROCOS framework are the visualisation tool RViz and the Gazebo simulator (the latter is not a component of ROS, but it is widely used in that ecosystem). Another potential candidate would be the rosbag, that allows to store and replay logged data, along with rqt tooling.

ROS is developed in a distribution fashion following the Ubuntu lifecycle (i.e. LTS and normal distributions). In the scope of ESROCOS the supported version of ROS will be “Kinetic Kame” which follows the Ubuntu 16.04 LTS version. Both will provide support till May 2021. Alternatively, and depending on the requirements of other components of the framework, such as TASTE, and other OGs, the version “Indigo Igloo” which follows Ubuntu 14.04 LTS could be used.

Gazebo is developed on a yearly basis and released, usually, every January. Current version is 8.0, and supports the Ubuntu 16.04 LTS. Its end-of-life is on 25/01/2019. In the scope of ESROCOS versions 8.0 and 9.0 should be supported (or version 7.1 if ROS “Indigo Igloo” is finally used).

In order to integrate RViz and Gazebo, there will be a need to create ASN.1 types for all the ROS messages to be used by the simulation plugins (for instance a camera plugin simulating a camera on a robot would publish the required images using a ROS specific message format). Equivalently for RViz one would have to generate ROS messages that needs to be visualized (e.g. the navigation data of a mobile platform can be shown with an odometry message). Using these message types one could create the required bridge components that would connect existing or newly generated TASTE models with ROS (see section 5.6.1.5).

5.5.4. PUS CONSOLE

This component is intended to monitor, debug and test the sending and reception of TC/TM from the Ground Segment, emulated by a control workstation (Linux PC).

Figure 5-46. PUS_CONSOLE preliminary design

It will contain a GUI (Graphical User Interface) where the operator can manually request TCs implemented by an application developed with ESROCOS and visualize the TM generated by the PUS_SERVICES as consequence of TC sent from GS or other on-board application components.

Further detail about its design will be provided for CDR stage.
5.6. INTEGRATION OF LEGACY SOFTWARE

5.6.1. MIDDLEWARE BRIDGES

5.6.1.1. DESCRIPTION OF THE APPROACH

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 5-47. TASTE-ROCK bridge](image1)

![Figure 5-48. TASTE-ROS bridge](image2)

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.

In the same way as for the interface wrapper approach, the design of the bridges must detail:

- The mapping of the abstractions of the respective middleware (e.g. components, message interfaces, temporal properties).
- The mapping of the data types.
- The definition of the software interfaces (e.g. function callbacks) to be used by the bridge.
- The process for the generation of the bridge component, which may combine manual and automated steps. The process must be integrated with both the TASTE and the external framework build processes.
The limitations of the mappings (e.g. constraints imposed in the models, non-supported features).

It must be taken into account that the architecture of a TASTE system must be static, i.e. it is not possible to dynamically define and connect provided or required interfaces. In consequence, each TASTE model will have its particular bridge component that forwards the provided and required interfaces needed by the model.

In the scope of the laboratory target, the following approaches are supported:

1. TASTE components as shared library, later embedded as ROS or ROCK components
2. TASTE components interfaces exported in a Python proxy

5.6.1.2. TASTE COMPONENTS AS SHARED LIBRARY

In its current design, TASTE models are transformed as a set of Linux (laboratory target) or RTEMS (space target) binaries. A first option to integrate TASTE components in an ESROCOS setting is to add an option to turn TASTE components into shared library that can be later loaded as ROS or ROCK components.

Building a shared library from a TASTE model requires two steps

1. Build a shared library out of a TASTE-CV model. This is a minor adaptation from the existing build mechanism that is already implemented;
2. Provides a user-level API to manipulate ports and interact with TASTE components. In its current implementation, only a low-level API is provided, allowing the user to interact with the TASTE-CV entities. This API should be abstracted to provide TASTE-CV concepts.

Step 1 is under the control of the Ocarina toolset. Step 2 would require adaptations of the vertical transformation implemented in buildsupport, maintained by ESA.

This approach enables the communication between the two middleware with limited overhead and in a way transparent to the user. However, it requires integrating the application lifecycle and build process of TASTE and the external framework.

5.6.1.3. TASTE COMPONENTS AS PYTHON MODULES

TASTE has a mode of operation where PI/RI are exported and visible through a proxy that is implemented a Python module. This mode of operation is coupled with a GUI that acts as a graphical front-end to interact with the model (see Figure 5-50).
The TASTE GUIs rely on POSIX message queues and contain a generated Python API to interface with the PI and RI code in the TASTE application. This infrastructure can be directly used to enable communication with an external middleware.

**Figure 5-50. Example of TASTE GUI**

This option is implementing as part of the existing vertical transformation. A set of Python accessors methods are built to interact with the PI/RIs. In its current form, two message queues are used to support:

- GUI: to support communication with the graphical user interface generated from the PI/RIs
- Regression testing: an API is generated to interact with the PI/RIs with additional services to wait for a specific message, or any message, trigger a timeout message, etc.

This Python API can be leveraged to provide interaction between TASTE and ROS, using Python as an implementation language for the TASTE to ROS/ROCK bridge. The main advantage of this approach is that it relies on functionality already in place in TASTE. Nevertheless, the integration with the application lifecycle and build process of the external frameworks needs to be assessed.

**5.6.1.4. TASTE-ROCK BRIDGE**

In ROCK data types are defined as C++ types. In ROCK’s toolchain, the C++ types are transformed into a XML data tree and eventually into an implementation independent
CORBA IDL files as part of ROCK’s toolchain. TASTE uses ASN.1 to define data types, and generates plain data structures in C or Ada for their usage in application code. It could be investigated if a more general tool that converts between IDL and ASN.1 files exists or could be developed with reasonable effort and would be useful for the ROCK-TASTE bridge.

In addition to the transformation of the data types, the construction of a bridge component requires a mapping of the communication interfaces defined by the two middleware environments.

The communication interfaces in TASTE are defined in the Interface View. The Table 5-3 presents the proposed mapping between this view and the elements provided by Orocos.

<table>
<thead>
<tr>
<th>TASTE IV</th>
<th>Orocos</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>None</td>
<td>Components have been replaced with nested functions in TASTE 2.0.</td>
</tr>
</tbody>
</table>
| Function     | Task context                | Each function is characterized by its interfaces, properties and internal state.  
|              |                             | In TASTE the activation behaviour is defined at interface level (a function may combine cyclic and sporadic interfaces), while in Orocos it is defined at task level.  
|              |                             | The file-driven activation has no equivalent in TASTE.                 |
| Sporadic     | Port (port-driven task)     | They provide asynchronous messages.                                     
| interface    |                             | TASTE interfaces may have multiple parameters, which must be grouped in order to transform to a single ROCK port data type. Also, they may have no parameters (event notification). |
| Cyclic       | Periodic task               | The periodic behaviour is local to a function or task, and does not reflect a communication between tasks. Therefore it does not make sense to map it to a bridge. |
| interface    |                             |                                                                         |
| Synchronous  | ~ Operation                 | The closest equivalent to a TASTE synchronous interface is an operation.    
| interface    |                             | On the TASTE side, synchronous interfaces are only accessible for tasks that reside in the same node. If the bridge is implemented in TASTE through a deployment view node that generates a shared library, it would not be possible to map synchronous interfaces over the bridge. For this reasons, it seems preferable not to support synchronous interfaces over the bridge. |
| Context      | Property                    | Context parameters in TASTE are similar to properties in ROCK. They are local to functions/tasks and do not take part in the communication, so they do not need to be mapped to the bridge interfaces. In any case, context parameters or properties could be used in the bridge for configuration purposes. |
| parameter    |                             |                                                                         |

The bridge is represented in the TASTE model as a function in the TASTE Interface View, and a corresponding task context in RTT.

The bridge function must have static interfaces with known data types defined in ASN.1. Depending on the application, the data that needs to be transferred between the TASTE and ROCK domains will be different. In consequence, the bridge component must be specific to each application. ESROCOS provides the tools needed to generate the bridge.

The structure of the bridge is the following:

- The bridge is a software component that behaves as a function from the point of view of TASTE, and as a task context from the point of view of ROCK.
- TASTE sporadic required interfaces are mapped to output ports in ROCK. For each mapped interface, the TASTE function has a sporadic provided interface that, when invoked, writes a message to the corresponding output port of the ROCK task context.

- ROCK input ports are mapped to TASTE sporadic provided interfaces. For each mapped interface, the ROCK task context has an input port that, when written, sends a message through a sporadic required interface of the TASTE function.

- The messages transferred by the mapped interfaces are converted from ASN.1 in the TASTE side to C++ in the ROCK side. ESROCOS provides an ASN.1 implementation of all the ROCK base types, as well as conversion functions between C and C++.

- The developer may create additional ASN.1 types and conversion functions to transfer other types of messages not covered by the robotics data types.

Protected and unprotected TASTE interfaces, and ROCK operations, are not transferred by the bridge.

5.6.1.5. TASTE-ROS BRIDGE

ROS is a middleware that allows multiple nodes to communicate in a synchronous and asynchronous manner using a message passing system. In order to create a bridge component that communicates the ROS and TASTE environment, a mapping of the concepts defined by both frameworks has been established.

Table 5-4. Mapping between TASTE Interface View and ROS elements

<table>
<thead>
<tr>
<th>TASTE IV</th>
<th>ROS</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>-</td>
<td>Components have been replaced with nested functions in TASTE 2.0.</td>
</tr>
<tr>
<td>Function</td>
<td>Node/Nodelet</td>
<td>Each function is characterized by its interfaces, properties and internal state. In TASTE the activation behaviour is defined at interface level (a function may combine cyclic and sporadic interfaces). In ROS the node has its internal state that operates at a specific rate. It can have timers to trigger events in a synchronous manner.</td>
</tr>
<tr>
<td>Sporadic interface</td>
<td>Publisher/Subscriber</td>
<td>They provide asynchronous messages. In ROS, subscribers can be used to trigger a behaviour from another node or TASTE element. Publishers can be used to trigger events from this node to other nodes or TASTE elements. TASTE interfaces may have multiple parameters, which must be grouped in order to transform to a single ROS message data type. Also, they may have no parameters (event notification). This is supported by the ROS empty message type.</td>
</tr>
<tr>
<td>Cyclic interface</td>
<td>ROS timer with callback</td>
<td>The periodic behaviour is local to a function or task, and does not reflect a communication between tasks. Therefore it does not make sense to map it to a bridge.</td>
</tr>
<tr>
<td>Synchronous interface (protected, unprotected)</td>
<td>ROS service</td>
<td>The closest equivalent to a TASTE synchronous interface is a service. In ROS services can be called by any node. The user must provide the code to call a specific service from his node.</td>
</tr>
<tr>
<td>Context parameter</td>
<td>Local variables</td>
<td>Parameters in ROS are local variables. They can be updated if code is provided to do so from the ROS parameter server. In any case, context parameters can be written in the ROS parameter server from the bridge for configuration purposes.</td>
</tr>
</tbody>
</table>

ROS elements can be directly mapped to TASTE elements. This could be done automatically by parsing the source file that defines a ROS node for instance and generating the equivalent interface in TASTE. ROS publishers and subscribers are equivalent to interfaces, while a TASTE function is equivalent of a ROS node. These concepts can be used to create the required bridge interfaces that will allow the TASTE models to communicate with ROS.
The bridge component will act as a translator between the two middleware. For this reason it will have a dual nature of being both a TASTE function and a ROS node.

To be able to communicate with the TASTE model, the bridge will be designed having requested and provided interfaces based on that. For each provided interface of the TASTE model the bridge will have a requested interface, likewise, for each requested interface of the TASTE model the bridge will have a provided interface. All these interfaces will use the same ASN.1 data types. ROS messages are composed of primitive types are defined in the following table.

<table>
<thead>
<tr>
<th>ROS Primitive Type</th>
<th>Serialization</th>
<th>C++</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>unsigned 8-bit int</td>
<td>uint8_t</td>
<td>bool</td>
</tr>
<tr>
<td>int8</td>
<td>signed 8-bit int</td>
<td>int8_t</td>
<td>int</td>
</tr>
<tr>
<td>uint8</td>
<td>unsigned 8-bit int</td>
<td>uint8_t</td>
<td>int</td>
</tr>
<tr>
<td>int16</td>
<td>signed 16-bit int</td>
<td>int16_t</td>
<td>int</td>
</tr>
<tr>
<td>uint16</td>
<td>unsigned 16-bit int</td>
<td>uint16_t</td>
<td>int</td>
</tr>
<tr>
<td>int32</td>
<td>signed 32-bit int</td>
<td>int32_t</td>
<td>int</td>
</tr>
<tr>
<td>uint32</td>
<td>unsigned 32-bit int</td>
<td>uint32_t</td>
<td>int</td>
</tr>
<tr>
<td>int64</td>
<td>signed 64-bit int</td>
<td>int64_t</td>
<td>long</td>
</tr>
<tr>
<td>uint64</td>
<td>unsigned 64-bit int</td>
<td>uint64_t</td>
<td>long</td>
</tr>
<tr>
<td>float32</td>
<td>32-bit IEEE float</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>float64</td>
<td>64-bit IEEE float</td>
<td>double</td>
<td>float</td>
</tr>
<tr>
<td>string</td>
<td>ascii string (4)</td>
<td>std::string</td>
<td>str</td>
</tr>
<tr>
<td>time</td>
<td>secs/nsecs unsigned 32-bit ints</td>
<td>ros::Time</td>
<td>rospy.Time</td>
</tr>
<tr>
<td>duration</td>
<td>secs/nsecs signed 32-bit ints</td>
<td>ros::Duration</td>
<td>rospy.Duration</td>
</tr>
</tbody>
</table>

In addition to the primitive types ROS also supports arrays. Details are presented in the following table.

<table>
<thead>
<tr>
<th>Array Type</th>
<th>Serialization</th>
<th>C++</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed-length</td>
<td>no extra serialization</td>
<td>0.11+: boost::array, else std::vector</td>
<td>tuple</td>
</tr>
<tr>
<td>variable-length</td>
<td>Uint32 length prefix</td>
<td>std::vector</td>
<td>tuple</td>
</tr>
<tr>
<td>uint8[]</td>
<td>see above</td>
<td>as above</td>
<td>bytes</td>
</tr>
<tr>
<td>bool[]</td>
<td>see above</td>
<td>std::vector&lt;uint8_t&gt;</td>
<td>list of bool</td>
</tr>
</tbody>
</table>

It can be easily seen that each ROS message can be translated to a struct generated by an ASN.1 compiler. The only issue would be the variable length lists allowed by ROS. For that a maximum list length could be specified that would be documented in the ESROCOS specifications and that will have to be respected by anyone wanting a ROS element to be integrated with TASTE, and therefore the whole ESROCOS software. This can be enforced at the ROS side by providing custom allocators to the message declaration that would allow variable length arrays of a maximum size and throw an error if more space is requested.

For the ROS nature of the bridge component message types are required to be equivalent with the ASN.1 data types. These message types will be used in the publishers and subscribers created to communicate with the ROS middleware. For each requested interface in the TASTE model a publisher will be created, while for each provided interface there will be a subscriber. If the TASTE component provides a synchronous interface the bridge will have a corresponding service that will link with that synchronous interface.
The bridge of each model would allow the communication between a TASTE generated model and a ROS element.  

A translator for ASN.1 types defined in the data view of the model can generate ROS message types. These messages will then be used by the ROS toolchain to generate the required C++ and python bindings provided by ROS.  

A bridging component can be then automatically generated by examining the TASTE model It must generate the correct interfaces for the bridging component, a required interface for the equivalent provided interface from the TASTE model and vice versa. Additionally, it must generate the equivalent ROS publishers and subscribers given the TASTE model. In that way whenever the bridge receives something in its provided interface it will publish the equivalent ROS message. In the same manner, whenever the ROS subscriber receives something, it will execute its callback and push the data in the corresponding bridge required interface.  

If just pushing the correctly converted data from a ROS message to ASN.1 is not enough, a user can manually insert code to achieve the required behaviour. This would be inserted in the corresponding callbacks of interfaces or subscribers.  

5.6.2. FRAMEWORK IMPORT TOOLS  

The ESROCOS framework reuses some existing software components from the ROCK and ROS ecosystems, and relies on tools and libraries to facilitate the integration of these components in TASTE.  

Together with the framework, a set of guidelines will be provided to help final users importing existing software built on ROS and ROCK for use in ESROCOS applications.  

In addition to these guidelines, the tools and libraries used in the development of the ESROCOS framework may be made available to the final user, so that application developers can use these tools and libraries to integrate their own legacy code.  

5.6.2.1. IMPORT TOOLS FOR ROCK  

In order to integrate a ROCK component (task context) in TASTE it is necessary to build a TASTE component (function) with equivalent interfaces. The definition of a ROCK component is provided in a .orogen file. This file defines the component interfaces (input and output ports), properties and activation logic (periodic, port-driven, etc.). The type of the data exchanged through the component interfaces is defined in C++ header files.  

The equivalent TASTE function is characterized by:

- A set of ASN.1 files that are to be used in the Data View of the TASTE application
- An Interface View for this component

The transformation does not consider the Concurrency, nor the Deployment View.  

The contents of the .orogen file in ROCK are mapped to an Interface View in TASTE as follows:  

Table 5-7. ROCK to TASTE elements mapping

<table>
<thead>
<tr>
<th>Element in ROCK</th>
<th>Element in TASTE</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>.orogen file</td>
<td>Interface view</td>
<td>An interface view is generated for each .orogen file. This interface view has the same name assigned to the .orogen component.</td>
</tr>
<tr>
<td>task_context</td>
<td>TASTE function</td>
<td>Each task context in ROCK is converted into a different function in TASTE. This function belongs to the container created for this .orogen file.</td>
</tr>
<tr>
<td>Element in ROCK</td>
<td>Element in TASTE</td>
<td>Comments</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------</td>
<td>----------</td>
</tr>
<tr>
<td>input_port</td>
<td>Provided interface</td>
<td>Each port in ROCK maps to a different provided interface in the Function associated to its context_task.</td>
</tr>
<tr>
<td>output_port</td>
<td>Required Interface</td>
<td>Each port in ROCK corresponds to a required interface defined in TASTE within the function associated to its context task.</td>
</tr>
<tr>
<td>property</td>
<td>Context parameters</td>
<td>Task properties are mapped to context parameters of each function. This has limitations, since these properties in rock can be modified, meanwhile in TASTE it is more difficult to modify them.</td>
</tr>
</tbody>
</table>

The SARGON project has developed the following elements, that will be the basis of the framework import tools in ESROCOS:

- The base robotics data types inherited from ROCK (see section 5.3.1) and the associated conversion functions.
- An RTT task adaptor library in C++ that provides stubs for certain RTT classes and methods, simplifying the integration of ROCK source code in TASTE functions.
- A tool to transform in a partially automated way a set of ROCK components (defined by an Orogen file and a set of C++ data types) into ASN.1 and AADL models for integration in TASTE.

The rock2taste tools partially automate the import process. Two different tools are available:

- A first one, ASN.1 Generator, generates the ASN.1 types based on the input C++ types.
- A second one, the most complete one, rock2taste Generator, generates an Interface View and the Data View from an existing .orogen file and its imported types.

The automatic generation of ASN.1 type definitions from C++ is useful, but has some limitations, namely:

- The rock2taste tool includes a C++ parser that performs the parsing of C++ headers in order to identify C++ structures, classes and typedefs. This parsing does not include a pre-processor, and therefore the code needs to be modified to allow the parsing. These are minimal changes, but require manual intervention for some specific cases.
- In addition, TASTE interface parameters must have a fixed size, so user intervention is required to translate common C++ types such as std::string or std::vector. To solve this problem, the tool allows to define a pre-defined transformation from a set of C++ types into ASN.1, based on a configuration file.
- C++ features such as templates, inheritance and deduced types are not supported by the tool.
- As mentioned earlier, in case it is necessary to reuse C++ classes, it is needed to make a conversion function to translate plain ASN.1 structures into C++ objects.
- The generation of the Interface view is not complete, properties are sometimes not properly generated, nor they are initialized.

The rock2taste tools will be used in the ESROCOS project to support the integration of exiting tools into the framework during development. The integration process requires still manual work.

The usage of rock2taste within the project will help identifying the possible improvements required to make the tools usable by a final user. Depending on the result of this trade-off, rock2taste will be made available as a tool within ESROCOS.
5.6.2.2. IMPORT TOOLS FOR ROS

The integration of an existing ROS component (nodes) in a TASTE component (function) can be done similarly to ROCK. In ROS, a message is a collection of primitive types. One can easily generate an equivalent ASN.1 type representing the same message structure. A ROS node can be mapped to a TASTE element as shown in the following table.

<table>
<thead>
<tr>
<th>Element in ROS</th>
<th>Element in TASTE</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node/Nodelet</td>
<td>TASTE function</td>
<td>Each node in ROS is converted into a different function in TASTE.</td>
</tr>
<tr>
<td>Subscriber</td>
<td>Provided Interface</td>
<td>Each subscriber in ROS maps to a different provided interface in the Function associated to its context_task</td>
</tr>
<tr>
<td>Publisher</td>
<td>Required Interface</td>
<td>Each publisher in ROS corresponds to a requested interface defined in TASTE within the function associated to its context_task</td>
</tr>
<tr>
<td>Service</td>
<td>Synchronous Interface</td>
<td>A service in ROS provides a synchronous method of communicating data, and potentially requesting some operation on them. The service call ends by a response transmitted to the caller.</td>
</tr>
<tr>
<td>Parameters</td>
<td>Context properties in TASTE</td>
<td>In ROS the local parameters of a node are nothing more than local variables. They can interact with the rest of the ROS deployment using the ROS provided parameter server. This allows for dynamic updates of the parameters.</td>
</tr>
</tbody>
</table>

As for ROCK, the import process may be supported by automation. In case that some tools are created to support the implementation of the ESROCOS framework, these will be considered for making them available to final users.

Following the same approach as for ROCK, ESROCOS will contain tools and libraries to facilitate the reuse of ROS components. These elements may be made available to the application developer. The following support may be provided:

- ASN.1 definitions for useful ROS types, and the associated conversion functions.
- Support library to simplify the integration of ROS components.
- A tool for the partially automated transformation of ROS types and interfaces into ASN.1 and AADL models.

5.6.3. FRAMEWORK EXPORT TOOLS

Although not strictly part of the workflow to develop robotics applications with ESROCOS, the framework provides tools to generate applications for ROCK and ROS from TASTE models (Interface and Data Views). This capability may help users of these frameworks to get acquainted with the tools offered by the ESROCOS framework and take advantage of some of the modelling and analysis capabilities it offers.

5.6.3.1. EXPORT FROM TASTE TO ROCK

The SARGON project developed a tool to export TASTE models to the ROCK framework. This tool, called taste2rock, will be updated and integrated in ESROCOS. 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 the SARGON laboratory environment, 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 transform maps each TASTE function to a ROCK task, and each interface to a ROCK port. The details of this mapping will be described below. In addition, the transform encapsulates the ASN.1 types, compiled to C, into C++ classes that can be used by RTT. The user can therefore use the ASN.1 types from the ROCK user code.

The tool takes as an input the Interface View model of a system defined in TASTE, and produces an equivalent set of ROCK components. The ASN.1 types defined in the TASTE Data View are wrapped in C++ classes usable by ROCK. However, ROCK has no equivalent of the TASTE Deployment and Concurrency Views.

The concept of deployment in ROCK refers to a set of components that are included together in a system, but it does not include any connectivity information or any mapping to system nodes. Instead, these aspects are handled directly by a start-up script coded in Ruby. The taste2rock tool can generate a simple deployment and start script for testing purposes. However, in order to perform a distributed requirement, a set of scripts need to be coded in Ruby to set up the exported components as needed.

As for the Concurrency View, it must be taken into account that ROCK uses a different middleware than TASTE and that this middleware does not provide any real-time guarantees. The settings defined in the Concurrency View cannot be mapped to ROCK. In addition, the analyses performed in the Concurrency View should be considered indicative only. To benefit from the runtime guarantees provided by TASTE, the software must be developed and run fully within the TASTE environment.

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

<table>
<thead>
<tr>
<th>TASTE IV element</th>
<th>ROCK transform</th>
<th>Remarks and limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>None</td>
<td>Components are helping elements that do not translate to runtime artefacts. In TASTE 2.0, components have been replaced by nested functions.</td>
</tr>
<tr>
<td>Function</td>
<td>Task context</td>
<td>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.</td>
</tr>
<tr>
<td>Sporadic PI</td>
<td>Input port + data type</td>
<td>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.</td>
</tr>
<tr>
<td>TASTE IV element</td>
<td>ROCK transform</td>
<td>Remarks and limitations</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Cyclic PI</td>
<td>Input port + data type + activator task</td>
<td>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.</td>
</tr>
<tr>
<td>Sporadic RI</td>
<td>Output port + data type</td>
<td>Each sporadic RI is mapped to an output port with the same data type as the matching PI.</td>
</tr>
<tr>
<td>Protected PI</td>
<td>Operation + data types</td>
<td>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.</td>
</tr>
<tr>
<td>Unprotected PI</td>
<td>Operation run by caller thread + data types</td>
<td>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.</td>
</tr>
<tr>
<td>Context parameter</td>
<td>Property + data type</td>
<td>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.</td>
</tr>
</tbody>
</table>

The taste2rock application is a Python application with simple procedural logic. Instead of describing the dynamic behaviour with a sequence diagram or a similar abstraction, this section focuses on the data flow of the transform.

The Figure 5-52 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.
- 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, optionally using 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 capability might not be made available in ESROCOS, as the interface with the TASTE function implementations depends on internal TASTE APIs and build files that may change.

**Figure 5-52. taste2rock transform logic**

5.6.3.2. EXPORT FROM TASTE TO ROS

Following the same approach as for taste2rock, the ESROCOS framework will provide a tool to generate ROS nodes from TASTE models. The next table provides a more detailed view of the mapping between the TASTE interface view and ROS elements.

**Table 5-9. Mapping between TASTE Interface View and ROS elements**

<table>
<thead>
<tr>
<th>TASTE IV</th>
<th>ROS</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>-</td>
<td>Components have been replaced with nested functions in TASTE 2.0.</td>
</tr>
<tr>
<td>TASTE IV</td>
<td>ROS</td>
<td>Remarks</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Function</td>
<td>Node/Nodelet</td>
<td>Each function is characterized by its interfaces, properties and internal state. In TASTE the activation behaviour is defined at interface level (a function may combine cyclic and sporadic interfaces). In ROS the node has its internal state that operates at a specific rate.</td>
</tr>
<tr>
<td>Sporadic</td>
<td>Publisher/Subscriber</td>
<td>In ROS, subscribers can be used to trigger a behaviour from another node or TASTE element. Publishers can be used to trigger events from this node to other nodes or TASTE elements. TASTE interfaces may have multiple parameters, which must be grouped in order to transform to a single ROS message data type. Also, they may have no parameters (event notification). This is supported by the ROS empty message type.</td>
</tr>
<tr>
<td>Cyclic</td>
<td>ROS timer with callback</td>
<td>ROS nodes can have timers to trigger events in a synchronous manner.</td>
</tr>
<tr>
<td>Synchronous</td>
<td>ROS service</td>
<td>The closest equivalent to a TASTE synchronous interface is a service. In ROS services can be called by any node. The user must provide the code to call a specific service from his node.</td>
</tr>
<tr>
<td>Context</td>
<td>Local variables</td>
<td>Parameters in ROS are local variables. They can be updated if code is provided to do so from the ROS parameter server. In any case, context parameters can be written in the ROS parameter server from the bridge for configuration purposes.</td>
</tr>
</tbody>
</table>
5.7. MANAGEMENT OF COMPONENT BUILD AND DEPENDENCIES

5.7.1. AUTOPROJ

Autoproj allows easily installing and maintaining software that is under source code form (usually from a version control system). It has been designed to support a package-oriented development process, where each package can have its own version control repository (think "distributed version control").

It provides the means for managing software packages of different types, coming from different sources to assemble and build robotics applications. Package types could be for instance components following a particular component model (e.g. AADL components), C++ libraries built with CMake or Makefile, Python libraries, etc.. Package source can be different Version Control Systems such as Git or SVN, Archive types such as ZIP or Tar, operating system dependency tools such as apt. Autoproj serves as the central tool handling all these packages, tracking dependencies between them and building/installing by interacting with the packages individual build systems.

![Figure 5-53. Package oriented software development](image)

Unlike the ROS build system, it is not bound to one build system, one VCS and one integration framework. The philosophy behind autoproj is:

- Supports both CMake and autotools, and can be adapted to other tools
- Supports different VCS: cvs, svn, git, plain tarballs.
- Software packages are plain packages, meaning that they can be built and installed /outside/ an autoproj tree, and are not tied at all to the autoproj build system.
- Leverage the actual OS package management system. Right now, only Debian-like systems (like Ubuntu) are supported, simply because it is the only one I have access to.
- Handle code generation properly

When applied to a software development framework for collaborative development work, one of the responsibilities of the framework’s development tools is take to take care of establishing a common workspace from the separated packages (cf. Figure 5-53). In collaborative development frameworks like ROS, ROCK or ESROCOS, three types of packages can be distinguished. Core packages are such packages that establish the infrastructure of the framework. In this category fall for example the build system, the component interface related libraries or code template generators, middleware and glue-code generators. Additional tools like logging tools, visualization tools or domain-specific software libraries/components can be considered as both core and contribution packages. Especially ROS showed the potential that lies in providing a mechanism for users to contribute to a growing contribution packages set. These are packages developed using the framework, that where released to public by the original developers (the users). The framework therefor must provide a workflow to publish the packages. The third category,
the user packages are those packages the user develops but are not made public. The framework development tools should support the incorporation of such user packages coming from previously unknown sources.

Besides the ability to scale with a growing number of functionalities that are implemented in separate packages, the package-oriented development workflow also provides means for maintaining this large number of packages by allowing maintenance on a per-package basis. Software is split into small chunks, where each of which should represent a single functionality or purpose within the framework. The individual packages can be each on its own be easier understood than a whole complex system could be, resulting in a larger potential group of contributors/maintainers. This increases software maintenance and improves feature tracking and testing.

![Diagram of package-oriented software development workflow](image)

**Figure 5-54. Package oriented software development - Implementation workflow**

As shown in Figure 5-54, the maintenance of the framework itself is rather a testing of integrated packages and providing feedback to the individual package development communities on an issues- or requirements-basis.
In an autoproj installation, developers share definitions for a set of packages (cf. Figure 5-55 top) that can depend on each other. Then, anyone can cherry-pick in these definitions to build its own installation (cf. Figure 5-55 bottom). In practice, one builds a complete configuration per-project).

Therefore the installation is subdivided into the following entities:

- Packages
- Package sets
- Buildconf

To setup a project, the project maintainer sets up a buildconf for his particular project. The buildconf includes package sets and selects the so called layout, a selection of individual packages.

The package sets are maintained by a group of content providers, e.g. a group of framework developers, or a research institute developing their own software packages. A package set itself is just a grouping for individual packages. The packages themselves must include the following information:

- How to get the package’s source code
- How to build the package
- On what the package depends. This can be either another package built by autoproj, or an operating system package

Also an identifier and classification is given to the package by its name (e.g. drivers/camera_usb_driver).

In addition to the management of the package dependencies and building with autoproj, it is foreseen to include other aspects of continuous integration for the ESROCOS framework. Features such as build reports, automated tests and code coverage statistics available may be enabled by the hosting infrastructure, e.g. GitHub. The scope of the continuous integration capabilities will be defined later in the project.

5.7.2.ESROCOS DEVELOPMENT SCRIPTS

ESROCOS will provide a set of scripts in order to facilitate the development workflow in the framework, and in particular the integration of TASTE model in the autoproj build infrastructure.

The framework will provide automation support for the following tasks:
- Creation of a new autoproj package containing a TASTE model.
- Build a TASTE model from autoproj.

The original TASTE development workflow follows a strict bottom-up approach for embedded system development approach:

- Init: In an initialization step, a workplace for your system is created
- Data Type Modelling: Data Types that will be used within the system are modelled
- Component Design/Implementation and Component Interconnection: The system is built from individual components, where each of which provides a component interface fulfilling a particular model. The component model allows for interconnections between components, by two different kinds of interfaces (calling and called interfaces) on the components. All components involved within a system are supposed to be identified and implemented and interconnected in this step of the workflow.
- Deployment: Modelling of runtime units (execution hardware) and busses and the assignment of individual components to the runtime units. In this step is defined which software parts run on which execution hardware and the matching compilers and glue code generators are triggered for preparing the executables.
- Analysis/Execution: In this step runtime characteristic of the modelled software can be evaluated, or the executables can be used within the target system.

One of the goals of ESROCOS is the opening of the workflow to groups of developers working collaborative together. Therefore the original TASTE workflow must be modified for supporting collaborative work. The most important aspects that are to be changed are summarized as follows:

- No more shared workspace for all software of the overall system, but individual workspaces for individual software parts.
- Separation of software development workflow and system integration workflow.
- Multi-project organization for better maintenance of software parts.
- Building a repository for collection software parts.

In Figure 5-56 we show the concept of the ESROCOS component development workflow and in Figure 5-57 of the ESROCOS system integration workflow. The workflows are orchestrated by what we call the ESROCOS Development Scripts, which build the user front-end to the software involved in the workflow.
Figure 5-56. ESROCOS Component Development Workflow
The new proposed workflow takes care about embedding software components into the autoproj infrastructure with the “Preinit” and “Publish” steps. For the component development workflow the “Data Type Modelling” and “Component Interconnection” steps become more and more irrelevant, since software components that are supposed to be used by others should use data types, that others also know (i.e. use base-types for their external interfaces) and it should not make as less assumptions as possible regarding its application context, such that interconnections with other components should be postponed until system integration. The main work for Component Developers is the Interface Design and Implementation, as well as the Publish step, if they want to contribute with their component to the overall component pool of the framework.

Figure 5-57. ESROCOS System Integration Workflow
For the System Integration workflow, the Component Interface Design and Implementation becomes obsolete, but the Component Interconnection and Deployment steps are the main work that is performed here.

The coordination of the development steps will be done by the ESROCOS Development Scripts. Currently we are developing mock-up implementations and test interfaces with the TASTE tools. The concrete design of the scripts will be defined later in the project.
6. TRACEABILITY BETWEEN REQUIREMENTS AND DESIGN

The Table 6-1 links each of the software products discussed in this document with the corresponding system requirements in D1.2 [AD.4]. When no such link exists, a justification is provided in *italics*.

**Table 6-1. Traceability between requirements and design**

<table>
<thead>
<tr>
<th>Activity</th>
<th>Component</th>
<th>Relevant PDD section</th>
<th>System Requirement(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model kinematic chains</td>
<td>Robot modelling tools</td>
<td>5.1</td>
<td>MOD-R01, MOD-R02, MOD-R03, MOD-R04, MOD-R05, MOD-R06, MOD-R07, MOD-R08, MOD-R09, MOD-R10, MOD-R11, MOD-R12</td>
</tr>
<tr>
<td>Model and analyse distributed real-time systems</td>
<td>TASTE</td>
<td>5.2.1</td>
<td>FDIR-01, FDIR-02, FDIR-03, FDIR-04, MID-R12, MID-R13, MID-R14, MID-R15, MID-R16, MID-R17, MID-R18, MID-R19, MID-R23, MID-R24, MID-R25, MID-R30, BIP-R02, TSP-R01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BIP-R03, BIP-R04</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BIP-R03, BIP-R04, BIP-R06, BIP-R10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BIP-R01, BIP-R03, BIP-R05, BIP-R07, BIP-R08, BIP-R09</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BIP-R03, BIP-R04, BIP-R10</td>
</tr>
<tr>
<td>Common robotics functions</td>
<td>Base robotics data types</td>
<td>5.3.1</td>
<td>DAT-R01, DAT-R02, DAT-R03, DAT-R04, DAT-R05, DAT-R06, DAT-R07</td>
</tr>
<tr>
<td></td>
<td>OpenCV</td>
<td>5.3.2</td>
<td>RLIB-R04</td>
</tr>
<tr>
<td></td>
<td>Eigen</td>
<td>5.3.3</td>
<td>RLIB-R04</td>
</tr>
<tr>
<td></td>
<td>Transformer</td>
<td>5.3.4</td>
<td>RLIB-R01</td>
</tr>
<tr>
<td></td>
<td>Stream aligner</td>
<td>5.3.5</td>
<td>RLIB-R03</td>
</tr>
<tr>
<td></td>
<td>PUS services</td>
<td>5.3.6</td>
<td>PUS-R01, PUS-R02, PUS-R03, PUS-R04, PUS-R05, PUS-R07, PUS-R08, PUS-R09, PUS-R10, PUS-R11, PUS-R12, PUS-R13, PUS-R14</td>
</tr>
<tr>
<td>Deploy and run</td>
<td>AIR</td>
<td>5.4.1</td>
<td>TSP-R02, TSP-R04, TSP-R06</td>
</tr>
<tr>
<td></td>
<td>HAIR</td>
<td>5.4.2</td>
<td>TSP-R05, TSP-R06</td>
</tr>
<tr>
<td></td>
<td>CAN bus driver</td>
<td>5.4.3</td>
<td>DRV-R01, DRV-R04, DRV-R05, DRV-R06, DRV-R12, DRV-R14, DRV-R15, DRV-R16</td>
</tr>
<tr>
<td></td>
<td>Ethernet driver</td>
<td>5.4.4</td>
<td>DRV-R07, DRV-R09, DRV-R11, DRV-R13, DRV-R14, DRV-R15, DRV-R16</td>
</tr>
<tr>
<td></td>
<td>SpaceWire driver</td>
<td>5.4.5</td>
<td><em>Added according to the SRR agreements with other OGs.</em></td>
</tr>
<tr>
<td></td>
<td>EtherCAT driver</td>
<td>5.4.4</td>
<td>DRV-R02, DRV-R07, DRV-R09, DRV-R11, DRV-R14, DRV-R15, DRV-R16</td>
</tr>
<tr>
<td>Monitor, debug, test</td>
<td>Data logger</td>
<td>5.5.1</td>
<td>VIS-R02, VIS-R03, VIS-R04</td>
</tr>
<tr>
<td></td>
<td>vizkit3d</td>
<td>5.5.2</td>
<td>VIS-R01</td>
</tr>
<tr>
<td></td>
<td>RVIZ</td>
<td>5.5.3</td>
<td>VIS-R01</td>
</tr>
<tr>
<td></td>
<td>Gazebo</td>
<td>5.5.3</td>
<td>INT-R03</td>
</tr>
<tr>
<td></td>
<td>PUS console</td>
<td>5.5.4</td>
<td>PUS-R14</td>
</tr>
<tr>
<td>Integrate legacy SW</td>
<td>Middleware bridges</td>
<td>5.6.1</td>
<td>INT-R01, INT-R05</td>
</tr>
<tr>
<td></td>
<td>Framework import tools</td>
<td>5.6.2</td>
<td>INT-R01, INT-R05</td>
</tr>
<tr>
<td></td>
<td>Framework export tools</td>
<td>5.6.3</td>
<td><em>Although not required for the purposes of ESROCOS, it is considered a useful capability to add to the framework.</em></td>
</tr>
<tr>
<td></td>
<td>Autoproj</td>
<td>5.7.1</td>
<td>CIN-R04.1, CIN-R04.2, CIN-R05, CIN-R06, CIN-R09, INT-R02</td>
</tr>
<tr>
<td>Activity</td>
<td>Component</td>
<td>Relevant PDD section</td>
<td>System Requirement(s)</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------</td>
<td>----------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>Manage build and dependencies</td>
<td>ESROCOS development scripts</td>
<td>5.7.2</td>
<td>MID-R30</td>
</tr>
</tbody>
</table>

The RCOS target requirements (TAR-RXX) apply to each software product according to the Lab/Space characterisation indicated in Table 6-1.