Due date of deliverable: 31-08-2018  
Start date of project: 01-11-2016  
Duration: 27 months  
Topic: COMPET-4-2016 Building Block b) Autonomy framework  
Time/Space/Resources planning and scheduling  
Work package: 4100, 4200, 4300, 4400, 4500, 4600 and 4700  
Manufacturing, Assembly, Integration and Testing  
(Task 4)  
Lead partner for this deliverable: GMV  

This project has received funding from the European Union’s Horizon 2020 Research and Innovation programme under Grant Agreement No 730086.  

<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: ERGO Team  
Approved by: GMV  
Authorized by: Jorge Ocón  
Code: ERGO_D4.3  
Version: 2.0  
Date: 10/01/2019  
Internal code: GMV 21809/18 V3/19
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>03/09/2018</td>
<td>113</td>
<td>First published version for ERGO TRR Milestone.</td>
</tr>
<tr>
<td>1.1</td>
<td>28/09/2018</td>
<td>115</td>
<td>Document updated after due to RIDs related to TRR milestone:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Fixed some typo errors in section 4.1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Re-done some links in sub-section of section 6.1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Completed section 8.1.2.3, which was empty.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Added a new section 8.1.2.5 for E4 files.</td>
</tr>
<tr>
<td>2.0</td>
<td>10/01/2019</td>
<td>116</td>
<td>Document updated to be online with last modifications performed in last project phase:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Added libssl-dev package dependency to section 8.2.1.2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Added information about Concurrency View in TASTE to section 6.1.2.3.2</td>
</tr>
</tbody>
</table>

**NOTICE**

The contents of this document are the copyright of the ERGO 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 ERGO Consortium. Such consent is hereby automatically given to the European Commission and PERASPERA PSA to use and disseminate.
# TABLE OF CONTENTS

**DOCUMENT STATUS SHEET** .......................................................................................................................... 2

**TABLE OF CONTENTS** ................................................................................................................................. 3

**LIST OF TABLES AND FIGURES** .................................................................................................................. 3

1. **INTRODUCTION** ........................................................................................................................................ 8
   1.1. **PURPOSE** ........................................................................................................................................... 10
   1.2. **SCOPE** ............................................................................................................................................ 10
   1.3. **CONTENTS** ....................................................................................................................................... 11

2. **REFERENCE AND APPLICABLE DOCUMENTS** ....................................................................................... 12
   2.1. **APPLICABLE DOCUMENTS** ............................................................................................................. 12
   2.2. **REFERENCE DOCUMENTS** ............................................................................................................. 12

3. **DEFINITIONS AND ACRONYMS** ............................................................................................................. 15
   3.1. **DEFINITIONS** .................................................................................................................................... 15
   3.2. **ACRONYMS** ...................................................................................................................................... 16

4. **FRAMEWORK PREPARATION** .................................................................................................................... 18
   4.1. **TASTE** .............................................................................................................................................. 18
      4.1.1. **TASTE installation** ....................................................................................................................... 18
           4.1.1.1. Installation over an official TASTE VM ....................................................................................... 18
           4.1.1.2. NATIVE installation on Linux 32/64 bits .................................................................................... 18
           4.1.1.3. Installation on a native Ubuntu trusty LAHR (14.04 LTS) ......................................................... 19
           4.1.1.4. Installation on a native ubuntu xenial (16.04 lts) ................................................................. 20
   4.2. **ERGO SW EXTERNAL DEPENDENCIES** ....................................................................................... 21

5. **USING THE FRAMEWORK** ........................................................................................................................ 22

6. **ARCHITECTURAL DESIGN** ....................................................................................................................... 24
   6.1. **TASTE MODELLING & SW BUILDING** .............................................................................................. 24
      6.1.1. **TASTE Modelling** ....................................................................................................................... 24
           6.1.1.1. Overview of the TASTE graphical editor ................................................................................... 25
           6.1.1.2. About TASTE AADL Models .................................................................................................... 26
           6.1.1.3. Start Modelling ......................................................................................................................... 27
           6.1.1.4. Data TYPES (Data view editor) ............................................................................................... 27
           6.1.1.5. Software Architecture (interface view editor) ......................................................................... 28
           6.1.1.6. hardware ARCHITECTURE (Deployment view editor) ....................................................... 37
           6.1.1.7. Timing Analysis (Concurrency View editor) ........................................................................... 46
      6.1.2. **TASTE SW Building** .................................................................................................................... 47
           6.1.2.1. Prerequisites .............................................................................................................................. 47
           6.1.2.2. Build process ............................................................................................................................. 47
           6.1.2.3. Build tailoring ........................................................................................................................... 47
   6.2. **V&V APPROACH & BIT TOOLS** ....................................................................................................... 48
      6.2.1. **V&V Approach** ............................................................................................................................ 48
      6.2.2. **FDIR Tool** ..................................................................................................................................... 49
      6.2.3. **iFinder** .......................................................................................................................................... 50
           6.2.3.1. Orbital scenario use case ........................................................................................................ 50
           6.2.3.2. Planetary scenario use case .................................................................................................... 53
7. PLANNING MODELLING (PDDL MODELS) ................................................................. 55
   7.1. REAL VALUES TO PDDL OBJECTS .................................................................... 56
8. AGENT DEVELOPMENT ......................................................................................... 57
   8.1. REACTORS INSTANTIATION ............................................................................. 57
       8.1.1. Adding a new reactor .............................................................................. 57
           8.1.1.1. Define the nature of the reactor ...................................................... 57
           8.1.1.2. Define the timelines needed ............................................................ 57
           8.1.1.3. Code the reactor functionality ......................................................... 60
           8.1.1.4. Implement the reactor ..................................................................... 60
       8.1.2. GCI particularization and use ................................................................... 62
           8.1.2.1. Autonomy level telecommands ....................................................... 62
           8.1.2.2. E1: Telecommands ....................................................................... 62
           8.1.2.3. E2: Timetag telecommands ............................................................. 63
           8.1.2.4. E3: OBCP .................................................................................... 63
           8.1.2.5. E4: High Level Goals ................................................................. 65
       8.1.3. CMD Reactors Particularization ................................................................. 67
8.2. GODA COMPONENT ............................................................................................ 68
   8.2.1. GODA Tailoring ......................................................................................... 69
       8.2.1.1. Directory structure .......................................................................... 69
       8.2.1.2. System requirements and installation .............................................. 69
8.3. MISSION PLANNER ............................................................................................ 71
   8.3.1. Reactor ....................................................................................................... 71
       8.3.1.1. Directory structure .......................................................................... 71
       8.3.1.2. System requirements and installation .............................................. 72
     8.3.2. Stellar ...................................................................................................... 72
       8.3.2.1. Directory structure .......................................................................... 73
       8.3.2.2. System requirements and installation .............................................. 73
9. FUNCTIONAL LAYER DEVELOPMENT ........................................................................ 74
   9.1. ROVER GUIDANCE .......................................................................................... 74
       9.1.1. Real-Time visualisation tool ................................................................. 75
9.2. ROBOTIC ARM TAILORING ............................................................................... 76
9.3. ADDING ADDITIONAL LIBRARIES .................................................................... 77
10. AGENT & FUNCTIONAL LAYER INTEGRATION ....................................................... 78
   10.1. AGENT CONTROLLER INTEGRATION - GENERAL REMARKS .................. 78
       10.1.1. Interfaces ............................................................................................. 78
       10.1.2. Data types ............................................................................................ 79
       10.1.3. Automatic code generation ................................................................. 80
11. APPLICATION TAILORING .................................................................................... 82
   11.1. AGENT CONFIGURATION .......................................................................... 82
       11.1.1. Agent Main Controller & Reactors Configuration ................................ 82
       11.1.2. GODA Configuration .......................................................................... 84
           11.1.2.1. General Configuration ................................................................. 84
           11.1.2.2. Mapping Configuration File .......................................................... 87
           11.1.2.3. GODA Execution Example ............................................................ 88
       11.1.3. Mission Planner Configuration ............................................................. 91
11.1.3.1. Reactor Configuration ................................................................. 91
11.1.3.2. Stellar Configuration ................................................................. 92

11.2. FUNCTIONAL LAYER CONFIGURATION ................................................. 92
11.2.1. Guidance Configuration ............................................................... 93
11.2.2. Robotic Arm Configuration ......................................................... 93
11.2.3. FDIR Configuration ..................................................................... 94

ANNEX A. ERGO PLANETARY USE CASE ................................................... 102
   A.1. ERGO PLANETARY CASE IMPLEMENTATION ...................................... 103
       A.1.1. TREX components – Deliberative and executive layer ................... 104
   A.2. EXAMPLE: ERGO_PLANETARY_CASE.CFG ........................................ 105
   A.3. EXAMPLE: PLANETARY_TIMELINE_CONFIG.CFG ................................ 108

ANNEX B. ERGO ORBITAL USE CASE .......................................................... 111
   B.1. ERGO ORBITAL CASE IMPLEMENTATION ........................................ 112
       B.1.1. TREX components – Deliberative and executive layer ................... 112
   B.2. EXAMPLE: ERGO_ORBITAL_CASE.CFG ........................................... 113
   B.3. EXAMPLE: ORBITAL_TIMELINE_CONFIG.CFG .................................. 114
# LIST OF TABLES AND FIGURES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2-1</td>
<td>Applicable documents</td>
<td>12</td>
</tr>
<tr>
<td>Table 2-2</td>
<td>Reference documents</td>
<td>12</td>
</tr>
<tr>
<td>Table 3-1</td>
<td>Definitions</td>
<td>15</td>
</tr>
<tr>
<td>Table 3-2</td>
<td>Acronyms</td>
<td>16</td>
</tr>
<tr>
<td>Table 4-1</td>
<td>ERGO SW external dependencies for scenarios</td>
<td>21</td>
</tr>
<tr>
<td>Table 4-2</td>
<td>Standard SW packages (install.sh)</td>
<td>21</td>
</tr>
<tr>
<td>Table 8-1</td>
<td>Parameter types available</td>
<td>59</td>
</tr>
<tr>
<td>Table 8-2</td>
<td>Methods called by the agent for this reactor to be functional</td>
<td>61</td>
</tr>
<tr>
<td>Table 8-3</td>
<td>Methods of Teleoreactor to be overloaded / defined when creating a new reactor</td>
<td>61</td>
</tr>
<tr>
<td>Table 8-4</td>
<td>TC E1 Example file</td>
<td>63</td>
</tr>
<tr>
<td>Table 8-5</td>
<td>TC E2 Example file</td>
<td>63</td>
</tr>
<tr>
<td>Table 8-6</td>
<td>Generic and example of a TC_E3_OBCP.dat file</td>
<td>64</td>
</tr>
<tr>
<td>Table 8-7</td>
<td>Available interface functions OBCP-ERGO</td>
<td>64</td>
</tr>
<tr>
<td>Table 8-8</td>
<td>Generic TC E4 file</td>
<td>66</td>
</tr>
<tr>
<td>Table 8-9</td>
<td>Example of a TC E4 file</td>
<td>66</td>
</tr>
<tr>
<td>Table 9-1</td>
<td>Example of RG log output</td>
<td>75</td>
</tr>
<tr>
<td>Table 9-2</td>
<td>Colour coding for map visualisation tool</td>
<td>76</td>
</tr>
<tr>
<td>Table 11-1</td>
<td>Example of the GODA general configuration file</td>
<td>85</td>
</tr>
<tr>
<td>Table 11-2</td>
<td>The <code>&lt;GoalGeneratorConfiguration&gt;</code> XML element</td>
<td>86</td>
</tr>
<tr>
<td>Table 11-3</td>
<td>The <code>&lt;Model&gt;</code> XML element of the <code>&lt;ClassifierConfiguration&gt;</code></td>
<td>86</td>
</tr>
<tr>
<td>Table 11-4</td>
<td>The <code>&lt;SaliencyConfiguration&gt;</code> XML element</td>
<td>86</td>
</tr>
<tr>
<td>Table 11-5</td>
<td>The <code>&lt;TestDataConfiguration&gt;</code> XML element</td>
<td>87</td>
</tr>
<tr>
<td>Table 11-6</td>
<td>Description of the Goal Generator mapping file fields</td>
<td>87</td>
</tr>
<tr>
<td>Table 11-7</td>
<td>Encoding for goal selection in the mapping file of the Goal Generator</td>
<td>87</td>
</tr>
<tr>
<td>Table 11-8</td>
<td>Mapping examples between labels and goals</td>
<td>88</td>
</tr>
<tr>
<td>Table 11-9</td>
<td>Example of a Mission Planner Reactor configuration file</td>
<td>91</td>
</tr>
<tr>
<td>Table 11-10</td>
<td>Example configuration</td>
<td>93</td>
</tr>
<tr>
<td>Table 11-11</td>
<td>Summary of the deliberative layer TREX components for ERGO Planetary case</td>
<td>104</td>
</tr>
<tr>
<td>Table 11-12</td>
<td>Summary of the executive layer TREX components for ERGO Planetary Case</td>
<td>105</td>
</tr>
<tr>
<td>Table 11-13</td>
<td>Relation between timelines and TASTE components</td>
<td>105</td>
</tr>
<tr>
<td>Table 11-14</td>
<td>Example of Agent’s reactor configuration for the planetary scenario</td>
<td>105</td>
</tr>
<tr>
<td>Table 11-15</td>
<td>Example of Reactor’s timelines configuration for the planetary scenario</td>
<td>108</td>
</tr>
<tr>
<td>Table 11-16</td>
<td>Summary of the deliberative layer TREX components for ERGO Orbital case</td>
<td>112</td>
</tr>
<tr>
<td>Table 11-17</td>
<td>Summary of the executive layer TREX components for ERGO Orbital Case</td>
<td>113</td>
</tr>
<tr>
<td>Table 11-18</td>
<td>Relation between timelines and TASTE components</td>
<td>113</td>
</tr>
<tr>
<td>Table 11-19</td>
<td>Example of Agent’s reactor configuration for the orbital scenario</td>
<td>113</td>
</tr>
<tr>
<td>Table 11-20</td>
<td>Example of Reactor’s timelines configuration for the orbital scenario</td>
<td>114</td>
</tr>
</tbody>
</table>
Figure 1-1: ERGO framework SW packages .................................8
Figure 5-1: Instantiating ERGO for a robotic platform .........................22
Figure 6-1: The TASTE workflow ...........................................24
Figure 6-2: TASTE Graphical Editor ...........................................26
Figure 6-3: V&V approach with the BIP Tools ................................49
Figure 6-4: Running the FDIR tool within Eclipse ...............................50
Figure 8-1: XML generic timeline configuration ................................58
Figure 8-2: XML example Timeline configuration file ............................60
Figure 8-3: XML generic configuration of a reactor .............................61
Figure 8-4: XML example reactor configuration .................................62
Figure 8-5: OBEP Example file for the planetary case ............................65
Figure 8-6 CMDReactor Timeline types .......................................67
Figure 8-7: GODA design overview ............................................68
Figure 8-8: Mission Planner components .......................................71
Figure 8-9: Stellar Design Overview ............................................72
Figure 9-1: RG directory structure .............................................74
Figure 9-2 Example representing the output of real time visualisation tool ........76
Figure 10-1 Example of RI and PI between TASTE and TREX components ....................78
Figure 10-2 Relation between TASTE interfaces, timelines and CMDReactors ............79
Figure 10-3: XML configuration of a timeline ................................80
Figure 10-4: ANS1 data structures and enums based on a timeline ...............80
Figure 11-1: Agent’s Configuration File .......................................84
Figure 11-2: Example input image ..............................................89
Figure 11-3: Expected Saliency output .........................................90
Figure 11-4: Expected segmented regions ......................................90
Figure 11-5: Expected bounding boxes .......................................91
Figure 11-6: Using configuration files for the FDIR component ................95
Figure 11-7: Interface and code for TASTE - BIP Engine connection of Battery port ..........97
Figure 11-8: Connection between BIP Engine and TASTE .....................97
Figure 11-9: Communication and dependency diagram for a FDIR component .................................98
Figure 11-10: Interface template for the TASTE – BIP model integration ......................98
Figure 11-11: The Sherpa TT robot (courtesy DFKI) ................................102
Figure 11-12: ERGO framework – planetary scenario ............................102
Figure 11-13 TASTE Interface view of ERGO Planetary case .....................104
Figure 11-14: Orbital use case ................................................111
Figure 11-15: ERGO orbital components .......................................111
Figure 11-16 TASTE Interface view of ERGO Orbital case ........................112
1. INTRODUCTION

This document is the ERGO User Manual for the European Robotic Goal-Oriented Autonomous Controller (ERGO).

ERGO (http://www.h2020-ergo.eu/) is one of the six space robotic projects in the frame of the PERASPERA SRC (2016 call) (http://www.h2020-peraspera.eu/). Its goal is to provide an Autonomy Framework capable of operating at different levels of autonomy, from tele-operations to full on-board autonomy.

The ERGO framework provides a set of packages and components that can be reused and tailored to develop a robotic platform that requires a high level of autonomy. The main packages and their components of the ERGO framework are shown in the following package diagram.

![Figure 1-1: ERGO framework SW packages]

The ERGO Framework is composed of four main SW packages:

- **SW1 ERGO CORE framework**

   ERGO is a SW framework for autonomy in space robots. It is aimed to build robotic applications in which a high level of autonomy is required. In particular, ERGO provides a set of functionalities that are available for building these applications.

   These functionalities consist of:

   - **Agent**: This component provides a generic robotic controller. Robotic controllers are the main component of a robotic architecture. Traditionally, robotic controllers are aimed to three-layer architectures. Our robotic controller provides an N-layered architecture, in which different control loops (reactive or deliberative) can be embedded into functional blocks that use a common interface. This is explained in Section 4.1 of [AD.12] document.
- **Ground Control interface**: A main component of any robotic application is the Ground Control interface. In ERGO the Ground control interface is in charge of handing Telecommands and Telemetry, and also is in charge of services provided for different levels of autonomy (time-tag commanding for E2, Action-Event service for E3, etc...). As part of the ERGO Framework, we provide a library that can be used for interfacing with ground. This is explained in Section 4.2 of [AD.12] document.

- **Stellar Mission Planner**: In ERGO we have developed a new planner, specifically designed taking into consideration the requirements for space applications. It uses PDDL, a standard planning language. Stellar provides a planner aimed to be used on board for space robotics applications. This is explained in Section 4.3 of [AD.12] document.

- **BIP Tools**: In order to provide formal verification of the system and to help in the development of an FDIR for any robotic application, two different tools have been developed by Verimag. Section 4.4 of [AD.12] document provides the design for these tools, namely:
  - **iFinder**: this tool consists of a method and implementation to generate invariants for timed systems and to verify safety requirements
  - **FDIR BIP tool**: A tool that automatically generates from a system design, recovery strategies and requirements to ensure an FDIR component.

- **TASTE Extensions**: These are the extensions to TASTE that are being performed in the frame of ERGO and which are required for TASTE in the frame of PERASPERA (in particular, as a contribution within ERGO to OG1). Further detail about the improvement to be done are provided in section 9 of [AD.12] document.

### SW2 ERGO SPECIFIC COMPONENTS framework

ERGO specific components framework is composed by a set of subcomponent with a specific functionality whose main features are:

- **Goal Oriented Data Analysis (GODA)**: is in charge of detecting serendipitous events, autonomously posting new goals to the planner whenever an interesting event is raised. It processes data from the perception system and generates new candidate goals as input to re-planning activities. This is explained in section 5.1 of [AD.12] document.

  Described functionality needs to be integrated with ERGO Agent through a reactor called GODA_Reactor (see section 6.1.2 of [AD.12]). This reactor will be part of the planetary configuration only.

- **Rover Guidance (RG)**: performing navigation map calculation, short and long-term path planning, resources estimation, hazard prevention and trajectory control. This component performs the computation of the path for the rover locomotion. It is described in section 5.2 of [AD.12] document.

  Described functionality needs to be integrated with ERGO Agent through a reactor called RG_Reactor (see section 6.1.4.1 of [AD.12] document). This reactor will be part of the planetary configuration only.

- **Robotic Arm (RARM)**: is in charge of planning & execution of the movements of the robotic arm. It plans trajectories and paths between points without any collision. This is described in section 5.3 of [AD.12] document.

  Described functionality needs to be integrated with ERGO Agent through a reactor called P-RAMP_Reactor in planetary scenario (see section 6.1.4.2 of [AD.12] document) and O-RAMP_Reactor in orbital scenario (see section 7.1.3.1 of [AD.12] document). This reactor will be tailored to be part of the planetary and orbital uses cases.
1.1. PURPOSE

This document describes how to instantiate and configure the ERGO framework in order to be used for future missions.

The work expected to be done by a user of ERGO will be limited to the instantiation of the software framework (in terms of required modelling, planning) with respect to its specifying mission scenario.

The main inputs to this document are the System Requirements [AD.8], the Interface Control Document [AD.9] and the ERGO Final Design Report [AD.12].

1.2. SCOPE

This document is one of the main outcome deliverable (D4.3) of the following Work-Packages of the ERGO major activity “WP 4000: Manufacturing, Assembly, Integration and Testing”:

- **WP4100** – ERGO Autonomous Controller Development (Leader: GMV, contributions from ET).
- **WP4200** – ERGO Planner Development (Leader: BAS, contributions from KCL, ADS, SCI, and GUK).
- **WP4300** – ERGO Executive Development (Leader: GMV, contributions from GUK and UGA).
- **WP4400** – ERGO Functional Layer Development (Leader: ET, contributions from UGA, GMV).
- **WP4500** – ERGO Formal V&V Development (Leader: UGA, contributions from GMV).
WP4600 – Common Demonstration Scenarios Development (Leader: GMV, contributions from ALL).
WP4700 – ERGO System Integration & Testing (Leader: GMV, contributions from ALL).

1.3. CONTENTS

This document is structured as follows:

- **Section 1.** This section presents the purpose, scope and structure of the document.
- **Section 2.** Lists other documents that complement or are needed to understand this document.
- **Section 3.** Defines terms and acronyms used in the document.
- **Section 4.** Describes how to prepare the ERGO framework environment before its usage.
- **Section 5.** Provides an overview about how the framework should be used.
- **Section 6.** Describes the major architecture design key drivers of ERGO framework.
- **Section 7.** Describes the modelling process and generation of PDDL files.
- **Section 8.** Describes how to use and instantiate the Agent major component.
- **Section 9.** Describes how to use and instantiate the Functional Layer major component.
- **Section 10.** Describes the integration process to be done between Agent and Functional Layer ERGO major components.
- **Section 11.** Describes the tailoring and configuration processes to be done after all ERGO major components (Agent and Functional Layer) have been developed and/or instantiated.
- **ANNEX A.** Provides a configuration example of ERGO for the planetary use case.
- **ANNEX B.** Provides a configuration example of ERGO for the orbital use case.
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.8]</td>
<td>ERGO D1.2 – System Requirements (v1.2)</td>
<td>Mar. 22nd, 2018</td>
</tr>
<tr>
<td>[AD.9]</td>
<td>ERGO D1.3 – Interface Control Document (v2.2)</td>
<td>Sep. 15th, 2017</td>
</tr>
<tr>
<td>[AD.10]</td>
<td>ERGO D2.1 – Preliminary Design Document (v1.3)</td>
<td>Mar. 22nd, 2018</td>
</tr>
<tr>
<td>[AD.11]</td>
<td>ERGO D2.2 – Testing and Integration Plan (v2.0)</td>
<td>Jan. 10th, 2019</td>
</tr>
<tr>
<td>[AD.12]</td>
<td>ERGO D3.1 – ERGO Final Design report (v2.0)</td>
<td>Jan. 10th, 2019</td>
</tr>
<tr>
<td>[AD.14]</td>
<td>ERGO D3.3 – Software Configuration File (v2.0)</td>
<td>Sep. 7th, 2018</td>
</tr>
</tbody>
</table>

2.2. REFERENCE DOCUMENTS

The following is the set of documents referenced:

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>[RD.1]</td>
<td>Onboard Adaptive Control of AUVs using Automated Planning and Execution</td>
</tr>
<tr>
<td>Ref.</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>[RD.12]</td>
<td>Fratini, A. Cesta and S. The timeline representation framework as a planning and scheduling software development environment. s.l. : 27th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG), 2008</td>
</tr>
<tr>
<td>[RD.18]</td>
<td>Pommerening, Florian and Helmert, Malte and Röger, Gabriele and Seipp, Jendrik. From non-Negative to General Operator Cost Partitioning. s.l. : AAAI, 2015</td>
</tr>
<tr>
<td>[RD.20]</td>
<td>Space Engineering; Spacecraft on-board control procedures [ECSS-E-ST-70-01C]. April 16, 2010</td>
</tr>
<tr>
<td>[RD.23]</td>
<td>Cushing, W., Kambhampati, S., Mausam, Weld, D.S.; When is temporal planning really temporal?; Proceedings of IJCAI 2007</td>
</tr>
<tr>
<td>[RD.30]</td>
<td>Open Motion planning library (OMPL)</td>
</tr>
<tr>
<td>Ref.</td>
<td>Title</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>[RD.37]</td>
<td>S.Bensalem, M.Bozga, TH.Nguyen, J.Sifakis: Compositional verification for component-based systems and application, IET Software</td>
</tr>
<tr>
<td>[RD.40]</td>
<td><a href="https://www.cs.umd.edu/projects/omega">https://www.cs.umd.edu/projects/omega</a></td>
</tr>
<tr>
<td>[RD.43]</td>
<td>ESROCOS Detailed Design D3.1</td>
</tr>
<tr>
<td>[RD.50]</td>
<td>Coles, Andrew and Fox, Maria and Long, Derek and Smith, Amanda: Planning with Problems Requiring Temporal Coordination: AAAI 2008</td>
</tr>
<tr>
<td>[RD.51]</td>
<td>Coles, Amanda and Coles, Andrew and Fox, Maria and Long, Derek: Forward-Chaining Partial-Order Planning: ICAPS 2010</td>
</tr>
</tbody>
</table>
3. DEFINITIONS AND ACRONYMS

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>Planning</td>
<td>Planning is the reasoning side of acting that aims to organize actions according to their expected outcomes in order to achieve some given goal. Automated Planning is the area of Artificial Intelligence (AI) that studies this process. One informal division in the field of automated planning can be done between classical-based and timelines-based planning, even though there are other approaches. Classical planning focuses to a great extent on performance and is based on broadly accepted standards such as PDDL. On the other hand, timeline-based planners are more expressive, especially in terms of temporal representation, but they do not share any common standard as they use to be proprietary products.</td>
</tr>
<tr>
<td>Goal</td>
<td>A goal specifies an action or state desired to be achieved by the target agent in the future. The planner’s task is to find a valid sequence of actions/states (the plan) that achieves those goals from a given initial state. There are four main properties that characterize goals: - Formal representation: Both classical and timelines-based planning use predicate logic to represent goals. - Temporal scope: In classical planning, goals must be satisfied at the end of the plan. On the other hand, goals in timeline planning can be defined to be achieved at any time within the temporal scope of the problem. - Hierarchy: In those systems modeled hierarchically, goals are classified as complex (high-level) and primitive (low-level). Complex goals must be decomposed (at planning or execution time) before they can be executed. - Hard/Soft: Hard goals must be achieved in the plan, while soft goals represent preferences that might be disregarded in the plan.</td>
</tr>
<tr>
<td>Domain</td>
<td>Formal description of the system to be controlled from the planning point of view. Different planning techniques use different modeling approaches. In the case of classical planning, the model consists of a description of the aspects of the world that are relevant for deliberation, along with a description of the actions the system is able to perform. These actions are described in terms of their conditions (when can they be applied), their duration and how they expect to change the current state of the world (when they are executed). In the case of timeline-based planning, the model consists of a set of components and a set of relevant physical constraints that influence the possible temporal evolution of such components (e.g., possible state transitions over time of the component, coordination constraints among different components, maximal capacity of resources, etc.).</td>
</tr>
<tr>
<td>Problem</td>
<td>Formal description of a planning task for a given domain. It defines the initial state of the world (the current state for each component of the domain in timeline-based planning), a list of goals (defined in timeline-based planning as states expected to be achieved by some of the components in the future) and a metric used to determine the plan quality. It is an input to the deliberative layer</td>
</tr>
<tr>
<td>State</td>
<td>The state at any given time is represented by the set of known facts about the world that are relevant for planning. In the case of timeline-based planning, it can also be seen as the current configuration of the state-machines used to model the system. In the case of classical planning it represents the facts that are known to be true at that time point.</td>
</tr>
<tr>
<td>Reactor</td>
<td>In our architecture, the concept of a reactor, is a separate part of the control architecture in charge of a control loop. Each control loop is embodied in a reactor that encapsulates all details of how to accomplish its control objectives.</td>
</tr>
<tr>
<td>Agent</td>
<td>Component that is responsible of harmonizing and coordinating the execution of the reactors. The controller has a common interface with all the reactors, using common, simple and well-defined interfaces among them. It follows a &quot;divide-and-conquer&quot; strategy that eases the minimization of resources and the dependencies among control loops</td>
</tr>
<tr>
<td>Latency</td>
<td>The latency of the reactor is the worst-case number of ticks that is allowed to deliberate over a request</td>
</tr>
<tr>
<td>Deliberation horizon</td>
<td>It represents the planning horizon of the reactor, quantifying the look-ahead for deliberation.</td>
</tr>
<tr>
<td>Execution frontier</td>
<td>It expresses the boundary between the past and the future at a given instant</td>
</tr>
<tr>
<td>Timeline</td>
<td>A timeline is a sequence of tokens</td>
</tr>
<tr>
<td>Token</td>
<td>Temporally qualified assertions expressed as a predicate with start and end time bounds defining the temporal scope over which they hold.</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>AAAI</td>
<td>Conference of the Association for the Advancement for Artificial Intelligence</td>
</tr>
<tr>
<td>AIJ</td>
<td>Artificial Intelligence (Journal)</td>
</tr>
<tr>
<td>APSI</td>
<td>Advanced Planning and Scheduling Initiative</td>
</tr>
<tr>
<td>ASTRA</td>
<td>Symposium on Advanced Space Technologies in Robotics and Automation</td>
</tr>
<tr>
<td>BIP</td>
<td>Behaviours, Interactions and Priorities</td>
</tr>
<tr>
<td>CDR</td>
<td>Critical Design Review</td>
</tr>
<tr>
<td>CFI</td>
<td>Customer Furnished Item</td>
</tr>
<tr>
<td>DDL</td>
<td>Domain Description Language</td>
</tr>
<tr>
<td>ECAI</td>
<td>European Conference on Artificial Intelligence</td>
</tr>
<tr>
<td>ECSS</td>
<td>European Cooperation for Space Standardization</td>
</tr>
<tr>
<td>ET</td>
<td>Ellidiss Technologies</td>
</tr>
<tr>
<td>FDIR</td>
<td>Fault, Diagnosis, Isolation and Recovery</td>
</tr>
<tr>
<td>GOAC</td>
<td>Goal Oriented Autonomous Controller</td>
</tr>
<tr>
<td>GODA</td>
<td>Goal Oriented Data Analysis component</td>
</tr>
<tr>
<td>GOTCHA</td>
<td>GOAC TRL Increase Convenience Enhancements Hardening and Application Extension</td>
</tr>
<tr>
<td>GUK</td>
<td>GMV United Kingdom</td>
</tr>
<tr>
<td>ICAPS</td>
<td>International Conference on Automated Planning and Scheduling</td>
</tr>
<tr>
<td>ICRA</td>
<td>IEEE International Conference on Robotics and Automation</td>
</tr>
<tr>
<td>IJCAI</td>
<td>International Joint Conference on Artificial Intelligence</td>
</tr>
<tr>
<td>IJRR</td>
<td>International Journal of Robotics Research</td>
</tr>
<tr>
<td>i-SAIRAS</td>
<td>International Symposium on Artificial Intelligence, Robotics and Automation in Space</td>
</tr>
<tr>
<td>IWPSS</td>
<td>International Workshop on Planning &amp; Scheduling for Space</td>
</tr>
<tr>
<td>JAIR</td>
<td>Journal of Artificial Intelligence Research</td>
</tr>
<tr>
<td>JFR</td>
<td>Journal of Field Robotics</td>
</tr>
<tr>
<td>KOM</td>
<td>Kick-off Meeting</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-driven Architecture</td>
</tr>
<tr>
<td>MER</td>
<td>Mars Exploration Rovers</td>
</tr>
<tr>
<td>MMOPS</td>
<td>Mars Mission On-board Planner and Scheduler</td>
</tr>
<tr>
<td>OBC</td>
<td>On-Board Computer</td>
</tr>
<tr>
<td>OBCP</td>
<td>On-Board Control Procedure</td>
</tr>
<tr>
<td>PDDL</td>
<td>Planning Domain Definition Language</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>PTU</td>
<td>Pan Tilt Unit</td>
</tr>
<tr>
<td>PUS</td>
<td>Packet Utilization Standards</td>
</tr>
<tr>
<td>RAM</td>
<td>IEEE Robotics and Automation Magazine</td>
</tr>
<tr>
<td>RCOS</td>
<td>Robot Control Operating System</td>
</tr>
<tr>
<td>ROI</td>
<td>Region Of Interest</td>
</tr>
<tr>
<td>ROS</td>
<td>Robot Operating System</td>
</tr>
<tr>
<td>RTEMS</td>
<td>Real Time Executive for Multiprocessor System</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real Time Operating System</td>
</tr>
<tr>
<td>SARGON</td>
<td>Space Automation &amp; Robotics General Controller</td>
</tr>
<tr>
<td>SCR</td>
<td>Sampling Catching Rover</td>
</tr>
<tr>
<td>SCY</td>
<td>Scisys</td>
</tr>
<tr>
<td>SFR</td>
<td>Sampling Fetching Rover</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>SMT</td>
<td>Satisfiability Module Theories</td>
</tr>
<tr>
<td>STN</td>
<td>Simple Temporal Network</td>
</tr>
<tr>
<td>TASTE</td>
<td>The Assert Set of Tools for Engineering</td>
</tr>
<tr>
<td>TIL</td>
<td>Time Initial Literal</td>
</tr>
<tr>
<td>TIST</td>
<td>ACM Transactions on Intelligent Systems and Technology</td>
</tr>
<tr>
<td>T-REX</td>
<td>Teleo-Reactive Executive</td>
</tr>
<tr>
<td>TRF</td>
<td>Timeline-based Representation Framework</td>
</tr>
<tr>
<td>TRO</td>
<td>IEEE Transactions on Robotics</td>
</tr>
<tr>
<td>UC3M</td>
<td>Universidad Carlos III de Madrid</td>
</tr>
<tr>
<td>UGA</td>
<td>University Grenoble Alpes</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>Validation &amp; Verification</td>
</tr>
<tr>
<td>WBS</td>
<td>Work Breakdown Structure</td>
</tr>
<tr>
<td>WPD</td>
<td>Work Package Description</td>
</tr>
</tbody>
</table>
4. FRAMEWORK PREPARATION

4.1. TASTE

TASTE is a software development framework composed of a set of interconnected tools and based on mature modelling languages such as ASN.1 for data structures, AADL for functional architectures and SDL for behaviours. This tool-chain includes graphical editors and code generators to automatically build software executables that preserve the real-time properties that are specified at design stage.

TASTE is specified, designed and maintained by the European Space Agency and a group of partners. The product is usually distributed within a dedicated Virtual Machine that can be downloaded freely from the TASTE website (http://taste.tools). However, extensions have been developed in the context of the ERGO (and ESROCOS) projects that may not have been encompassed within the standard distribution yet.

4.1.1. TASTE INSTALLATION

At the very beginning of the ERGO project, it was decided to create a specific ERGO version of TASTE editors in order to add any needed new features without conflicting with the official TASTE distribution. This was realized by creating an ERGO branch in the Ellidiss Technologies SVN repository. These ERGO specific TASTE editors are available on the Ellidiss Technologies wiki (http://www.ellidiss.fr/public/wiki/attachment/wiki/ErgoUpdatePage).

The remaining part of the TASTE tool suite was unchanged and can thus be obtained from the standard distribution channel. The ERGO-TASTE tool chain can be installed either by customizing an “official” TASTE Virtual Machine (VM), or by setting up a native distribution on the development platform. Both approaches are explained below.

4.1.1.1. INSTALLATION OVER AN OFFICIAL TASTE VM

In order to install the ERGO specific TASTE tool suite over an official VM, first install Virtualbox and then download the official VM (http://download.tuxfamily.org/taste/TASTE-VM-9.0.3-32bit.ova).

Add the TASTE VM to Virtualbox, launch it and log as user taste with password tastevm. Then go into the tool-src directory and launch the Update-TASTE.sh script in order to get the latest version of the TASTE tool suite from the ESA git repository.

Once the TASTE tool suite is up-to-date, get the ERGO specific version of the TASTE editors on the Ellidiss Technologies wiki (32 bits version) and decompresses it under /home/taste/ERGO-Editor.

To enable the use of the ERGO specific version of the TASTE editors, add /home/taste/ERGO-Editor/bin at the beginning of PATH environment variable.

4.1.1.2. NATIVE INSTALLATION ON LINUX 32/64 BITS

4.1.1.2.1. KNOWN ISSUES

- Under Ubuntu Trusty Tahr (14.04 LTS), the list of needed packages for TASTE installation contains unknown package versions, unknown packages and a gnat version conflict. Please follow the procedure described in 4.4.1.3 INSTALLATION ON A NATIVE UBUNTU TRUSTY LAHR (14.04 LTS) in that case.

- Under Ubuntu Xenial (16.04 LTS), the list of needed packages for TASTE installation contains unknown package versions and a gnat version conflict. Please follow the procedure described in 4.4.1.4 INSTALLATION ON A NATIVE UBUNTU XENIAL (16.04 LTS) in that case.

4.1.1.2.2. PREREQUISITES

The three following points are mandatory to do a native installation:

- /bin/sh shall points to /bin/bash (needed by the TASTE tool chain)
Git shall be installed on the system (needed by the online install to check out the TASTE tool chain from the ESA git repository)

- The current user shall have sudo rights

### 4.1.1.2.3. INSTALLATION

First, get the [installation script](http://www.h2020-ergo.eu/wp-admin/admin.php?page=sharedfiles&option=com_sharedfiles&action=download&dir=Work%2FOnlineInstaller&item=installErgo.sh&order=name&srt=yes) from the ERGO web site. Copy and launch it in the repository where the ERGO version of the TASTE tool chain shall be installed. A file named `.bashrc.ergo` is generated in the home directory of the current user. To use the ERGO specific version of the TASTE tool chain, add it to the `.bashrc` file of the current user.

### 4.1.1.2.4. UPDATE

To update the ERGO specific version of the TASTE tool chain, go into the `tool-src` directory belonging to the previously defined installation directory. Launching the `Update-ERGO.sh` script will update the tool chain.

### 4.1.1.3. INSTALLATION ON A NATIVE UBUNTU TRUSTY LAHR (14.04 LTS)

#### 4.1.1.3.1. PREREQUISITES

The following points are mandatory to use the Ubuntu 14.04 online installation:

- `/bin/sh` shall points to `/bin/bash` (needed by the TASTE tool chain)
- Git shall be installed on the system (needed by the online install to check out the TASTE tool chain from the ESA git repository)
- The current user shall have sudo rights
- GNAT 2017 shall be installed either globally or locally. In both cases, the bin directory of the GNAT installation shall be added to the PATH environment variable. GNAT 2017 can be downloaded from the [adacore website](https://www.adacore.com/download/more).

#### 4.1.1.3.2. INSTALLATION

First, get the [installation script](http://www.h2020-ergo.eu/wp-admin/admin.php?page=sharedfiles&option=com_sharedfiles&action=download&dir=Work%2FOnlineInstaller%2FUbuntu-14.04-LTS&item=installErgoForUbuntu-14.04-LTS.sh&order=name&srt=yes) from the ERGO web site. Copy and launch it in the repository where the ERGO version of the TASTE tool chain shall be installed. A file named `.bashrc.ergo` is generated in the home directory of the current user. To use the ERGO specific version of the TASTE tool chain, add it to the `.bashrc` file of the current user.

#### 4.1.1.3.3. UPDATE

To update the ERGO specific version of the TASTE tool chain, go into the `tool-src` directory belonging to the installation directory chosen in the previous chapter. Launching the `Update-ERGO.sh` script will update the tool chain.

#### 4.1.1.3.4. KNOWN ISSUES

- If Python raises errors about connection to the dbus, add the following line to your `.bashrc` environment file: `export NO_AT_BRIDGE=1`
- If git cannot clone for certificate reasons, open a terminal and execute the following command: `export GIT_SSL_NO_VERIFY=true`
If at update time git cannot update for certificate reasons, open a terminal, go into the tool-src directory belonging to the installation directory and execute the following command:

git config http.sslVerify "false"

### 4.1.1.4. INSTALLATION ON A NATIVE UBUNTU XENIAL (16.04 LTS)

Due to ESROCOS requirements, the ERGO specific version of the TASTE tool chain is based on the version 9.1 of the TASTE tool chain which is not public right now.

#### 4.1.1.4.1. PREREQUISITES

The following points are mandatory to use the Ubuntu 16.04 online installation:

- /bin/sh shall points to /bin/bash (needed by the TASTE tool chain)
- Git shall be installed on the system (needed by the online install to check out the TASTE tool chain from the ESA git repository)
- The current user shall have sudo rights

#### 4.1.1.4.2. INSTALLATION

First, get the [installation script](http://www.h2020-ergo.eu/wp-admin/admin.php?page=sharedfiles&option=com_sharedfiles&action=download&dir=Work%2F0.Taste%2FOnlineInstaller%2FUbuntu-16.04-LTS%2FWithTASTE-9.1&item=installErgoForUbuntu-16.04-LTS.sh&order=name&sr=yes) from the ERGO web site.

Copy and launch it in the repository the ERGO version of the TASTE tool chain shall be installed.

A file named `.bashrc.ergo` is generated in the home directory of the current user. To use the ERGO specific version of the TASTE tool chain, add it to the `.bashrc` file of the current user.

#### 4.1.1.4.3. UPDATE

To update the ERGO specific version of the TASTE tool chain, go into the tool-src directory belonging to the installation directory chosen in the previous chapter. Launching the `Update-ERGO.sh` script will update the tool chain.

#### 4.1.1.4.4. KNOWN ISSUES

If Python raises errors about connection to the dbus, add the following line to your `.bashrc` environment file:

```
export NO_AT_BRIDGE=1
```

If git cannot clone for certificate reasons, open a terminal and execute the following command:

```
export GIT_SSL_NO_VERIFY=true
```

If at update time git cannot update for certificate reasons, open a terminal, go into the tool-src directory belonging to the installation directory and execute the following command:

```
git config http.sslVerify "false"
```
4.2. ERGO SW EXTERNAL DEPENDENCIES

ERGO SW external dependencies have been uploaded to ERGO GitLab repo ([https://spass-git-ext.qmv.com/ERGO/ExternalDeps.git](https://spass-git-ext.qmv.com/ERGO/ExternalDeps.git)), where they can be downloaded before to be installed.

The following tables list the SW external dependencies for ERGO framework for planetary and orbital use cases.

### Table 4-1: ERGO SW external dependencies for scenarios

<table>
<thead>
<tr>
<th>External SW Dependency</th>
<th>Description</th>
<th>Applicable to scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>DART</td>
<td>Dynamic Animation and Robotics Toolkit. The library provides data structures and algorithms for kinematic and dynamic applications in robotics and computer animation. [DART website] - <a href="http://dartsim.github.io/">http://dartsim.github.io/</a></td>
<td>Planetary &amp; Orbital</td>
</tr>
<tr>
<td>FCL</td>
<td>The Flexible Collision Library. FCL is a library for performing three types of proximity queries on a pair of geometric models composed of triangles. [FCL Website]: <a href="https://travis-ci.org/flexible-collision-library/fcl">https://travis-ci.org/flexible-collision-library/fcl</a></td>
<td>Planetary &amp; Orbital</td>
</tr>
<tr>
<td>OMPL</td>
<td>The Open Motion Planning Library (OMPL). OMPL, the Open Motion Planning Library, consists of many state-of-the-art sampling-based motion planning algorithms. [OMPL Website]: <a href="https://travis-ci.org/ompl/ompl">https://travis-ci.org/ompl/ompl</a></td>
<td>Planetary &amp; Orbital</td>
</tr>
<tr>
<td>OpenCV</td>
<td>Open Source Computer Vision Library (OpenCV). OpenCV was designed for computational efficiency and with a strong focus on real-time applications. * Homepage: <a href="http://opencv.org">http://opencv.org</a> * Docs: <a href="http://docs.opencv.org/master/">http://docs.opencv.org/master/</a></td>
<td>Planetary &amp; Orbital</td>
</tr>
<tr>
<td>uEye</td>
<td>IDS - Imaging Development Systems GmbH (uEye Linux SDK installation).</td>
<td>Planetary &amp; Orbital</td>
</tr>
<tr>
<td>Eigen</td>
<td>Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.</td>
<td>Planetary &amp; Orbital</td>
</tr>
</tbody>
</table>

### Table 4-2: Standard SW packages (install.sh)

```bash
# Install comertial deps
sudo apt-get install jam -y
sudo apt install libeigen3-dev libccd-dev libassimp-dev libboost-regex-dev libtinyxml2-dev liburdfdom-dev libboost-program-options-dev libyaml-cpp-dev freeglut3-dev libxml2-dev
sudo ln -s /usr/include/eigen3/Eigen/ /usr/include/

# Needed for the GUI interface for the GCI
sudo apt-get install python3-tk

# Java to be able to use Vitre
sudo apt-get install default-jre

# Deps of the OBCP library
pip3 install jsonschema
pip3 install simplejson

sudo apt-get install libcunit1 libcunit1-dev
```
5. USING THE FRAMEWORK

ERGO is a general purpose controller architecture and, as such, it can be instantiated for any space robotic system.

The instantiation of ERGO is a process that involves a number of design and implementation tasks and design decisions. The overall process can be outlined as the following steps:

1. Development of the Functional layer.
2. Development of the Reactors that conform the agent.

The following figure describes in detail the process.

![Instantiating ERGO for a robotic platform](image)

**Figure 5-1: Instantiating ERGO for a robotic platform**

The instantiation of ERGO is a process that involves a number of design and implementation tasks and design decisions. ERGO relies on three different components that need to be tailored for each mission, the following activities are to be performed:

- **Architectural Design**: this is done via TASTE and described in section 6.

- **Planning Modelling**: this consists of the definition of the PDDL files that describe the planning model as well as an implementation of the external functions that are used in the domain. Further detail is provided in section 7.

- **Agent Development**: this consists of the definition of the reactors that will conform the agent. It contains the executive and the deliberative layers of our architecture. It is composed of reactors that share a common interface. Each reactor is responsible of a set of timelines (state variables) of the system. In the ERGO architecture, three different types of reactors are envisaged:

  - The Ground control interface reactor that can be developed by tailoring the Ground control interface services provided in SW1.
  - The Goda reactor which acts as a deliberative reactor into the agent and delegates its business logic in the GODA API component provided in SW2.
The Mission planner reactor that can be developed by tailoring the mission planner reactor provided in SW1 (containing Stellar, the mission planner).

A set of reactors that interface with the functional layer, that is, forward the commands of the executive and receive observations from the functional layer. These are the so-called “command dispatcher” reactors in the ERGO architecture.

**Functional Layer Development** built using a set of TASTE functions (software components), in which some of the components of SW2 can be used.

The ERGO agent is a single TASTE function (i.e. component) that interfaces with the functional layer (a set of TASTE functions) via specific application-interfaces which are also modelled with TASTE. From a TASTE model, code can be generated and deployed for different platforms (namely, Linux and RTEMS).

**Agent & Functional Layer Integration**

This process requires the generation of both executables based on TASTE. For a proper distribution of the workload, we recommend to deploy the agent into a single executable and the remaining functions of the functional layer into a separate executable. See the next section “Architectural design” for a guide on how to generate the executables.
6. ARCHITECTURAL DESIGN

6.1. TASTE MODELLING & SW BUILDING

6.1.1. TASTE MODELLING

Once the TASTE tool-chain has been properly installed onto the development platform, the next step consists in using the graphical editors to build a model of the future applicative software.

Taste models can be edited through the TASTE editor. It is a graphical tool that contains in facts four interconnected editors:

- The Data View editor: it contains the description of all the TASTE data types for the project (used in IV functions, context parameters and PI/RI parameters). These data types are described by ASN.1 and ACN representations.
- The Interface View (IV in short) editor: it is used to specify the software part of the project. An interface view model is hierarchical composition of Functions, each of them defining “Provided” and “Required” Interfaces that can be connected together.
- The Deployment View (DV in short) editor: it is used to specify the hardware part (execution platform) of the project. A deployment view project is composed of Nodes, Processors, Partitions, Devices and Buses. Interfaces of buses and devices can be connected together.
- The Concurrency View (CV in short) editor: it is used to analyse the timing behaviour of the model before generating code. Theoretical scheduling can be computed thanks to the Cheddar tool and real-time simulation can be performed thanks to the Marzhin AADL simulator.

The overall TASTE modelling workflow is shown in Figure 6-1.

![Figure 6-1: The TASTE workflow](image)
The TASTE editor uses various background technologies:

- **GMP (Graphic Model Processing):** generic graphical framework developed by Ellidiss Technologies.
- **LMP (Logic Model Processing):** generic model transformation and AADL toolbox developed by Ellidiss Technologies.
- **Cheddar** ([http://beru.univ-brest.fr/~singhoff/cheddar](http://beru.univ-brest.fr/~singhoff/cheddar)): real-time scheduling analysis tool developed by the University of Brest.
- **Marzhin:** AADL run-time simulator developed by Virtualys and Ellidiss Technologies.
- **Ocarina** ([http://www.openaadl.org/ocarina.html](http://www.openaadl.org/ocarina.html)): AADL compiler and code generator developed by ISAE.

### 6.1.1.1. OVERVIEW OF THE TASTE GRAPHICAL EDITOR

The TASTE graphical editor consists in a single window that encompasses:

- A main menu and button bar. The button bar is updated upon the current modelling or verification activity, as defined by the selection tab.
- A models browser where the overall project hierarchy and organization is displayed in a deployable tree structure. This browser may be hidden from the View main menu.
- A set of selection tabs to enable one of the proposed TASTE modelling or verification activities, and update the content of the working area.
- A working area, showing either:
  - A textual editor where the ASN.1 and ACN representations of a TASTE DataView model can be edited, or
  - A diagram editor where a graphical representation of a TASTE Interface View or Deployment View model can be edited, or
  - A read-only area showing the results of the timing analysis of a TASTE Concurrency View model, or
  - A read-only textual editor that displays the generated textual AADL code
- A log view where main editing operations are logged. This log view is hidden by default and can be made visible from the View main menu.
- A status bar showing system information, warning and error messages
6.1.1.2. ABOUT TASTE AADL MODELS

The TASTE models that can be loaded into and saved from the TASTE editor are serialised in AADL format. The AADL language is a SAE international standard that is used to model and analyse the architecture of safety critical real time systems. More information about AADL can be found at [http://www.aadl.info/](http://www.aadl.info/).

For the purpose of TASTE models serialisation, only a subset of the AADL language is used, and the TASTE entities are associated with specific AADL constructs that have been defined to ensure a correct semantic mapping. A TASTE project makes use of several kinds of models. Each of them is associated with a dedicated AADL subset. These models are:

- Data View: automatically generated from ASN.1 source code.
- Interface View: automatically generated from the Interface View editor.
- Hardware Library: provided by the Ocarina environment.
- Deployment View: automatically generated from the Deployment View editor.
- Concurrency View: automatically generated by the TASTE Build Support utility.
- Customised Properties: specified by the `TASTE_IV_Properties` and `TASTE_DV_Properties` files located in the `config` directory of the TASTE editor distribution.

All the AADL files that are involved in a TASTE project are either provided or automatically generated. It is thus not necessary – and even not recommended – for a TASTE end-user to edit the AADL models. Directly editing the provided or generated AADL files may lead to load errors and model corruption.
Within the TASTE editor, the AADL files can be loaded either in a generic way from the File main menu (Load), or for a particular kind of model from the contextual menus of the models browser (Load DataView, Load IV, Load DV, Load CV). The generic load process only works for AADL files that have been produced by version 2.0 or greater of the TASTE editor. Older files can be made compatible by adding a comment line at the beginning of the AADL file to specify which kind of TASTE model is concerned:

```plaintext
-- type dataview
-- type interfaceview
-- type deploymentview
-- type concurrencyview
-- type hwlibrary
```

Note that only one model of each kind can be loaded at a time. A new load action will replace the previously loaded model. When it is required to have access to several models of the same kind, the import menu must be used instead. This is in particular the case for reusing components from another Interface View model.

Also note that loading an IV model automatically loads the referenced DataView. Similarly, loading a DV model automatically loads the referenced HW Library and IV, as well as the DataView referenced by this IV. When a new DataView is explicitly or implicitly loaded, it is merged with the already loaded ones. This action may raise an error when the ASN.1 toolbox is not available (i.e. outside the TASTE VM), but does not prevent the DataView to be used for modelling activities.

### 6.1.1.3. START MODELLING

In the frame of the ERGO project, it will be assumed that TASTE editors are use in the TASTE tool chain environment, either using the virtual machine provided by ESA, or installing the full TASTE tool chain on a native development platform.

In the TASTE tool chain, several bash scripts are provided to ease the development process. Two of them are used to start modelling with the TASTE graphical editors either creating a brand new model or editing an existing model.

#### 6.1.1.3.1. CREATING A NEW PROJECT

To create a new model, the designer use the script named `taste-create-project` in a terminal. The target directory where to create the new model is then enquired. Simply press return to use the current directory.

The script creates the specified target directory if needed and opens the TASTE graphical editor. In the project directory, the following files are created:

- DataView.asn and DataView.acn which are copied from the TASTE environment, representing a default basic Dataview.
- DataView.aadl which is the AADL representation of the Dataview model
- InterfaceView.aadl representing the AADL serialization of an empty software model
- DeploymentView.aadl representing the AADL serialization of an empty hardware model

#### 6.1.1.3.2. EDITING AN EXISTING PROJECT

To edit an existing model, the designer use the script named `taste-edit-project` in a terminal. The script shall be started in the model root directory.

The script opens the TASTE graphical editor and loads the DataView, InterfaceView and DeploymentView from the current directory.

#### 6.1.1.4. DATA TYPES (DATA VIEW EDITOR)

The data view editor goal is to let the user load and edit the data types used within the project.

The DataView.aadl file is the AADL representation of the types defined in a set of ASN.1 and ACN files. The DataView.aadl file is generated either in command line using the update-data-view script provided in the TASTE tool chain, or using the TASTE Editor.
The user can manipulate taste Data View models through the following means:

- The main menu: load a data view with the entry "File→Load"
- The models browser contextual menu on DataView entry: load and unload data view

While loading a Data view, the ASN.1 files used to generate it are used to populate the DataView tree of the models browser. Each ASN.1 file is represented as a new branch and its type definitions are represented as leaves. If an ASN.1 file is related to an ACN file, the ACN file is automatically used to generate the Data View and as such is represented in the DataView tree of the models browser.

While double clicking on a type in the models browser, the Data View tab is raised in the working area. The displayed ASN.1 file is the file defining the selected type. Within this editor, the user can either edit, remove or add a type. When saving the ASN.1 file, the DataView.aadl file is automatically updated and re-loaded to take into account the up to date type definitions.

The current Data View can be populated with existing ASN.1 files using the "Import ASN file" entry in the model browser contextual menu from DataView. When importing an existing ASN.1 file, the Data View is updated by adding the imported ASN.1 file to the list of files used to generate the AADL code. The new Data View is then loaded. In a similar way, "Unload file" allows removing an ASN.1 file from the list of files used to generate the Data View which is updated and reloaded. If an ACN file related to the ASN.1 exists, it is imported/removed automatically.

The current Data View can also be initialized directly from the models browser using the "New ASN file" in the contextual menu from DataView. A new branch corresponding to the new ASN.1 file is created under the DataView tree. The opposite operation is "Unload file" in the contextual menu from the ASN.1 file to be unloaded. When unloading an ASN.1 file, the related ACN file is also unloaded, if any.

Finally, the current Data View can also be populated directly from the models browser using the "New ACN file" from the contextual menu from an ASN.1 file. A new node with the same name as the selected ASN.1 but with .acn extension is created beside the ASN.1 file. From the models browser, an ACN file cannot be unloaded, but it can be removed. Be careful, removing an ACN file from the Data View will also remove the ACN file on the file system.

6.1.1.5. SOFTWARE ARCHITECTURE (INTERFACE VIEW EDITOR)

The aim of the Interface view editor is to let the user build the functional design of the applicative software, using architectural constructs that enforce predefined real-time behaviours that will be preserved after code generation of the actual executable.

6.1.1.5.1. INITIALIZING AN IV MODEL

To initialize the modelling process with a blank Interface View, the designer can create a new Interface View with the:

- The “File→New” entry of the main menu
- The “New” entry of the toolbar
6.1.1.5.2. ADDING A NEW GRAPHICAL ELEMENT

The addition of a new graphical element to the current software model can be done through:

- The "New" entry of the main menu
- The toolbar of the main window
- The contextual menu of the diagram

6.1.1.5.3. TASTE FUNCTION

A TASTE function represents a software module in the target source code.

6.1.1.5.3.1 Adding a TASTE Function

To add a TASTE function to the model, use one of the three options described in the chapter "6.1.1.5.2 Adding a new graphical element". Once selected, the TASTE editor enters the TASTE Function creation mode.

While in TASTE Function creation mode, the first click on the Interface View diagram sets the upper left corner of the TASTE Function and the second one its lower right corner.

According to the TASTE modelling rules, the creation of a TASTE Function is legal if the Function fulfils a set of creation rules. These rules are checked on the fly by the TASTE graphical editor, based on the position of the mouse. If the rules are fulfilled, the mouse pointer becomes a cross icon otherwise it becomes a wrong-way icon. The corresponding message is displayed in the status bar of the editor.

6.1.1.5.3.2 Removing a TASTE Function

To remove a TASTE Function from the model, select the Function to remove then:

- Press the delete key on the keyboard
- In the "Edit" entry of the main menu, select the "Delete" entry
- In the toolbar, click on the "Delete" button

A TASTE Function can also be removed through its contextual menu, selecting the "Delete" item.
6.1.5.3.3 Editing the properties of a TASTE Function

To edit the properties of a TASTE Function, select the Function to edit then either:

- In the "Edit" entry of the main menu select the "Edit Properties" item
- In the toolbar, click on the "edit properties" button

Properties can also be edited either by double clicking on the Function to edit or by selecting the "Edit Properties" entry in the contextual menu of the Function to edit.

In edition mode, a popup dialog box named "Edit Data" is raised (image beside). The dialog box is composed of the following four tabs:

- "Function Attributes", to edit the direct attributes of the TASTE Function
- "Context Parameters", to create/edit/remove context parameters to be linked with the current TASTE Function
- "Report", to display the error/warning linked to the properties edition
- "Description", to set the description of the current TASTE Function in plain text format

The next paragraphs provide more details about the content of the main tabs.

**Function Attributes:**

- The "Label" field holds the name of the current function. While editing it, if the naming rules for TASTE Functions are not fulfilled, the string is displayed in red and the OK button remains inactive. The label can also be edited directly from the Interface View diagram by clicking on it.
- "Language" is a combo box to set the coding language to be used to implement the current Function
- "Source text" is a path to a zip file containing the code of the current Function. Since only terminal TASTE Functions are allowed to have code, if this field is not empty, no sub-function creation will be allowed in the current Function.
- if the "Constrained Mode" box is enabled, only protected and unprotected Provided Interfaces are allowed in the interface of the descendants of the current Function
- "Fill Color" allows customizing the display colour of the current Function on the diagram
- "Version" is a field used to set the version of the current Function
- "is_Component_Type" is a Boolean flag that must be set if the current Function shall be considered as a Function type. This is an experimental feature that works only with SDL for now. It shall not be used in the frame of ERGO.
- "is_instance_of" is a field to specify which Function Type the current Function is an instance of. If it is empty, it means that the current Function is a plain TASTE Function (i.e. a singleton). This is an experimental feature that works only with SDL for now. It shall not be used in the frame of ERGO.

**Context Parameters:**
A context parameter is a global variable in the scope of the current TASTE Function. A context parameter is defined by the three fields: name, type and default value. The three of them are mandatory, thus the OK button of the edit properties dialog box is not active until there are all set.

The “Name” field holds the name of the context parameter. When editing it, if the naming rules of the context parameter are not fulfilled, the string is displayed in red and the OK button is inactive. The “Type” field is a combo box populated thanks to the loaded Data View.

To add a context parameter, the user must click on the “+” button on the upper right corner.

To remove a context parameter, click on the “-” button on the right of the context parameter definition.

To see the ASN.1 definition of the context parameter click on the question mark on the right of the context parameter definition.

### 6.1.1.5.3.4 Reuse of TASTE Functions

TASTE Functions may be reused in two ways. They can be either duplicated within the scope of the same IV model or exported for further import in the scope of another IV model.

**Copy/Paste of TASTE Functions**

Using the TASTE graphical editor, it is possible to duplicate an existing TASTE Function from the current model by copying/pasting it. This TASTE Function may be a non terminal Function. In that case, the entire hierarchy and its internal connections will be copied too.

To copy a TASTE Function, select the Function to copy then:

- In the “Edit” entry of the main menu select the “Copy” entry
- In the toolbar, click on the “copy” button

A TASTE Function can also be copied through the Function contextual menu by selecting the “Copy” entry.

To paste a copy of the TASTE Function either select the “Paste” item in the “Edit” menu, or click on the Paste button in the toolbar. Then move the mouse in the diagram, in the area where the mouse pointer becomes a cross, one can click on the diagram to create the copy. The mouse coordinates will be the upper left corner of the new item.

The paste operation is also possible through the contextual menu of the Interface View diagram while selecting the “Paste” entry. The upper left corner of the created copy is then the position of the mouse when opening the contextual menu.

The user must be aware that only graphical information is copied. Even if code already exists for the original TASTE Function, the copied TASTE Function does not have any code related to it.

**Export TASTE Functions**

With the TASTE graphical editor, it is possible to create libraries of reusable components across different IV models. To export a TASTE Function from the current IV model, either select the TASTE Function and open the "File" entry of the main menu, or open the contextual menu of the TASTE Function in the diagram. In the opened menu, select the “Export Functions” entry.

A new Interface View with the exported TASTE Function as root element is created. First a file explorer is raised to set the directory to export the TASTE Function to. Then to serialize the TASTE Function into AADL format, the designer is prompted for the name of the new interface view through a popup dialog box.
In the targeted directory, a new InterfaceView.aadl file is created. This file is the serialization of the exported TASTE Function. The ASN.1 and ACN files from the original Interface View are also copied to the targeted directory and a new DataView.aadl file is compiled in the directory using these files. For each TASTE terminal Function being exported, if the property "Source Text" points to an existing file this file is copied into the targeted directory and the corresponding "Source Text" property in the new Interface View is updated.

Import of TASTE Functions

To import a TASTE Function into the current model, it must have been exported as an Interface View before. The import is done either through the "File" entry of the main menu by selecting the "Import Interface View" entry or through the contextual menu of InterfaceViews node in the models browser tree selecting the "Import Interface View" entry. A file explorer is raised to select the InterfaceView AADL file.

The imported Interface View is now visible in the models browser tree (see picture beside, Peek_Poke is an imported Interface View).

To import a TASTE Function from the imported Interface View into the current model, drag’n’drop the TASTE Function to import from the models browser to the Interface View diagram.

When importing a TASTE Function, the related DataView must also be imported. Thus, the coherence between the current DataView and the imported one is checked prior to merge them if possible. Otherwise an import error message is raised.

If an imported terminal TASTE Function has a non empty property "Source Text", the corresponding file is not copied to the current model repository. The source text still point to the original repository.

6.1.1.5.4. TASTE INTERFACES

Provided interfaces (PI in short) represent the functional services that are implemented within the local Function and can be called periodically or on demand from remote Functions. On the contrary, Required Interfaces (RI in short) are the local representation of a called service in a remote Function.

For client-server communications between sibling Functions, a given RI can be connected to a single PI but a given PI can be connected to several RIs. For the connection to be accepted, the characteristics of both interfaces must follow precise matching rules. These characteristics include the number and type of parameters and the way the service can be triggered (e.g. synchronously or asynchronously).

6.1.1.5.4.1 Adding a Provided Interface to a Function

To add a Provided Interface to a TASTE function of the model, use one of the tree possibilities described in the chapter "6.1.1.5.2 Adding a new graphical element". Once selected, the TASTE editor enters the Provided Interface creation mode.

While in Provided Interface creation mode, click on the TASTE Function to add a Provided Interface to it. The Provided Interface will be created within the bounding box of the Function at the nearest point where the user clicked.

According to the TASTE modelling rules, the creation of a Provided Interface is legal if the Provided Interface fulfils a set of creation rules. These rules are checked on the fly by the TASTE graphical editor, depending on the position of the mouse. If the rules are fulfilled, the mouse pointer becomes a cross icon otherwise it becomes a wrong-way icon. The corresponding message is displayed in the status bar of the editor.
When a Provided Interface is created, the properties edition dialog box is automatically popped up in order to set its name (see “Editing the properties of a Provided Interface” paragraph).

6.1.1.5.4.2 Removing a Provided Interface from a Function

To remove a Provided Interface from the model, select the Provided Interface to remove, then:

- Press the delete key of the keyboard
- In the “Edit” entry of the main menu, select the “Delete” entry
- In the toolbar, click on the “Delete” button

A Provided Interface can also be removed through its contextual menu, selecting the “Delete” entry.

6.1.1.5.4.3 Editing the properties of a Provided Interface

To edit the properties of a Provided Interface, select the Provided Interface to edit then:

- In the “Edit” entry of the main menu, select the “Edit Properties” entry
- In the toolbar, click on the “edit properties” button

Properties can also be edited either by double clicking on the Provided Interface to edit or by selecting the “Edit Properties” entry in the contextual menu of the Provided Interface to edit.

In edition mode, a popup dialog box named “Edit Data” is raised (image above). The dialog box is composed of the following five tabs:

- “PI Attributes”, is used to edit the direct attributes of the Provided Interface. This tab is dynamically modified depending on the kind of the Provided Interface (see detailed description of the tab below)
- “Parameters”, to add/remove parameters to the Provided Interface signature
- “MSC”, to create/edit/remove MSC files to be linked with the current Provided Interface
- “Report”, to display the error/warning linked to the properties edition
- “Description”, to edit the description of the current Provided Interface in plain text format

The next paragraphs provide more details about the content of the main tabs.

**PI Attributes:**

Within this tab, the following fields are available:

- “Operation Name” (for all kinds of PIs) is the name of the current Provided Interface. When editing it, if the validation rules of the Provided Interface naming are not fulfilled, the string is displayed in red and the OK button remains inactive. The label can also be edited directly from the Interface View diagram by clicking on it.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation Name</td>
<td>startFunc</td>
</tr>
<tr>
<td>Kind</td>
<td>sporadic</td>
</tr>
<tr>
<td>Min Inter-arrival Time (ms)</td>
<td>25</td>
</tr>
<tr>
<td>Deadline (ms)</td>
<td>250</td>
</tr>
<tr>
<td>WCET (ms)</td>
<td>150</td>
</tr>
<tr>
<td>Queue size</td>
<td>2</td>
</tr>
</tbody>
</table>

In edition mode, a popup dialog box named “Edit Data” is raised (image above). The dialog box is composed of the following five tabs:
“Kind” (for all kinds of PIs) set the kind of the Provided Interface using a combo box whose values are: cyclic, sporadic, protected and unprotected. Cyclic and sporadic interfaces are managed concurrently by the real-time scheduler. Cyclic interfaces are dispatched periodically by the system clock whereas sporadic interfaces are triggered by a remote RI. Protected and unprotected calls correspond to sequential calls. Unlike unprotected interfaces, protected interfaces prevent concurrent execution of several PIs of the same Function.

“Period” (for cyclic PIs only) is the period of the Provided Interface in milliseconds

“Min Inter-arrival Time” (for sporadic only) is the minimum inter-arrival time of the Provided Interface in milliseconds

“Deadline” (for all kinds of PIs) is the deadline of the Provided Interface in milliseconds

“WCET” (for all kinds of PIs) is the worst case execution time of the Provided Interface in milliseconds

“Queue size” (for sporadic only) is the length of the calling queue of the Provided Interface

**Parameters:**

This tab looks as shown below:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Encoding Protocol</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PI1</td>
<td>T_UInt32</td>
<td>NATIVE</td>
<td>IN</td>
</tr>
</tbody>
</table>

To add a parameter the user must click on the “+” button on the upper right corner.

To remove a parameter, click on the “−” button on the right of the parameter to remove.

- The “Name” field holds the name of the parameter. While editing it, if the naming rules of the parameter are not fulfilled, the string is displayed in red and the OK button remains inactive.
- The “Type” field is a combo box populated according to the data types provided by the loaded Data View.
- The “Encoding Protocol” field is used to specify how the parameter value will be encoded. Possible options are “NATIVE” when no specific transformation is required, “UPER” for compact binary encoding and “ACN” to use specific encoding rules specified in a .acn file.
- The “Direction” field may be “IN” or “OUT”.

**MSC:**

Message Sequence Charts files are used to define test scenario that applies to the selected Provided Interface.

To associate a MSC file to the Provided Interface, the user must click on the “+” button in the upper right corner of the properties dialog box.

On the right hand side of the MSC path field, there are three buttons:

- The first one is used to edit/create a MSC file using the TASTE tool chain MSC editor
- The middle one is used to load a MSC file
6.1.1.5.4.4 Adding a Required Interface to a Function

To add a Required Interface to a TASTE function, use one of the three possibilities described in the chapter "6.1.1.5.2 Adding a new graphical element". Once selected, the TASTE editor enters a Required Interface creation mode.

When in Required Interface creation mode, click on the TASTE Function to add a Required Interface to it. The Required Interface will be created within the bounding box of the Function at the nearest point where the user clicked.

According to the TASTE modelling rules, the creation of a Required Interface is legal if the Required Interface fulfils a set of creation rules. These rules are checked on the fly by the TASTE graphical editor, depending on the position of the mouse. If the rules are fulfilled, the mouse pointer becomes a cross icon otherwise it becomes a wrong-way icon. The corresponding message is also displayed in the status bar of the editor.

6.1.1.5.4.5 Removing a Required Interface from a Function

To remove a Required Interface from the Function, select the Required Interface to remove then:

- Press the delete key on the keyboard
- In the “Edit” entry of the main menu, select the “Delete” entry
- In the toolbar, click on the “Delete” button

A Required Interface can also be removed through its contextual menu, selecting the “Delete” entry.

6.1.1.5.4.6 Editing the properties of a Required Interface

To edit the properties of a Required Interface, select the Required Interface to edit then:

- In the “Edit” entry of the main menu select the "Edit Properties" entry
- In the toolbar, click on the “edit properties” button

Properties can also be edited either by double clicking on the Required Interface to edit or by selecting the "Edit Properties" entry in the contextual menu of the Required Interface to edit.

In edition mode, a popup dialog box named "Edit Data" is raised (image below). The dialog box is composed of the following four tabs:

- "RI Attributes" is used to edit the direct attributes of the Required Interface.
- "Parameters", to add/remove parameters to the Required Interface signature
- "Report", to display the error/warning linked to the properties edition
- "Description", to set the description of the current Required Interface in plain text format

The next paragraphs provide more details about the content of the main tabs.

RI Attributes:
The “Label” field holds the name of the current Required Interface. While editing it, if the naming rules of the Required Interface are not fulfilled, the string is displayed in red and the OK button remains inactive. The label can also be edited directly from the Interface View diagram by clicking onto it.

"Inherits from PI" is a Boolean field to set the connection and validation mode between Provided and Required Interfaces (see “Connection between Provided and Required Interfaces” paragraph)

“Kind” set the kind of the Required Interface using a combo box which values are: sporadic, protected, unprotected and any. When specified for a RI, this attribute acts as a requirement to restrict the allowed PIs to which it can be connected. When set to “any”, no such restriction will apply.

**Parameters:**

<table>
<thead>
<tr>
<th>RI Attributes</th>
<th>Parameters</th>
<th>Report</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Type</td>
<td>Encoding Protocol</td>
<td>Direction</td>
</tr>
<tr>
<td>p1</td>
<td>MyInteger</td>
<td>NATIVE</td>
<td>IN</td>
</tr>
</tbody>
</table>

To add a parameter, the user must click on the “+” button on the upper right corner. To remove a parameter, click on the “-” button on the right of the parameter to remove.

- The “Name” field holds the name of the parameter. While editing it, if the naming rules of the parameter are not fulfilled, the string is displayed in red and the OK button remains inactive.
- The “Type” field is a combo box populated according to the data types provided by the loaded Data View.
- The “Encoding Protocol” field is used to specify how the parameter value will be encoded. Possible options are “NATIVE” when no specific transformation is required, “UPER” for compact binary encoding and “ACN” to use specific encoding rules specified in a .acn file.
- The “Direction” field may be “IN” or “OUT”.

### 6.1.1.5.4.7 Connecting Interfaces

To connect two interfaces either drag and drop from the source Interface to the target Interface, or right click on the source Interface, select the “Connect” entry from the contextual menu then click on the targeted Interface.

Two Interfaces belonging to the same TASTE Function cannot be connected. Connections can express either client-server communication between sibling Functions (RI-PI) or delegation of service across hierarchical Functions (PI-PI or RI-RI).

**Connecting a Provided Interface and a Required Interface**

A cyclic Provided Interface cannot be connected to a Required Interface, as it is already implicitly connected to the system clock.

If the “Inherits from PI” attribute of the Required Interface is set to true, then the connection is always accepted. The kind and the parameters of the Required Interfaces are then overwritten by those of the connected Provided Interface.

If the “Inherits from PI” attribute of the Required Interface is set to false, then the connection is accepted only if the kind and the parameters of both Interfaces match.
Provided and Required Interfaces can be connected together only if they belong to sibling TASTE Functions. Nevertheless, the user can connect “Provided” and “Required” Interfaces from any hierarchical level. In that case the TASTE editor will manage the creation and connection of the intermediate “Provided” and “Required” Interfaces to create the requested end to end connection.

Connecting two Provided Interfaces or two Required Interfaces

While connecting two Interfaces between a parent and a child Function (or reverse), in order to express a delegation of service across hierarchical Functions, the target Interface attributes are overwritten with those of the source Interface of the connection. However, if the connection is drawn across several levels of hierarchy, the lower level Interface attributes are always used to overwrite those of all the other Interfaces contributing to the same delegation chain.

6.1.1.5.5. CODE MANAGEMENT

TASTE Functions must be associated to source code. This source code can be either automatically generated by separate tools like Matlab/Simulink or OpenGeode, or written by hand. In the latter case, it is possible to initialize the code skeleton so that it complies with the defined Interfaces.

6.1.1.5.5.1 Code skeletons generation

Once the model is complete, or each time the model is modified, the code skeletons shall be generated again prior to edit it.

To generate the code skeletons of the current IV model, either open the contextual menu in the diagram or open the “Tools” menu. From the opened menu select the “Generate code skeletons” item.

The TASTE tool chain code generation process creates for each terminal TASTE Function:

- A directory named against the TASTE Function name
- Code file(s) named against the TASTE Function name in the previously created directory

Depending on the source language specified for each TASTE Function, the code skeleton generation process will overwrite or not the existing files (see warning at the beginning of the generated files). For instance, if the language is C, the .h file is overwritten each time the code skeletons are generated whereas the .c file is generated only once.

6.1.1.5.5.2 Code edition

To add behavioural code into the generated skeletons, either open the contextual menu of the TASTE Function to edit or select the TASTE Function to edit and open the “Tools” menu. From the opened menu select the code editor corresponding to the language of the current TASTE Function.

6.1.1.6. HARDWARE ARCHITECTURE (DEPLOYMENT VIEW EDITOR)

The aim of the Deployment View editor is to define the architecture of the execution platform on to which the software specified by the Interface View will run. A DV model is composed of one or several Nodes that can be connected via Buses. A Node contains at least a Processor and may also include Devices that act as bus drivers. A Processor is composed of at least one Partition to which the logical Functions of the Interface View are bound.
Processor, Bus and Device components must instantiate one of the corresponding classifier components that are defined in the TASTE HW Library.

The nominal process for building a DV model consists in adding components from the HW Library, connecting Devices to Buses, and then allocating IV Functions to DV Partitions and IV Connections to DV Buses.

6.1.1.6.1. INITIALIZING A DV MODEL

To initialize the current model with a blank Deployment View, the designer can request the creation of a new Deployment View through:

- The “File→New” entry of the main menu
- The “New” entry of the toolbar
- The "New DV" entry of the contextual menu of the DeploymentView tree item in the models browser.

6.1.1.6.2. ADDING A NEW GRAPHICAL ELEMENT

In the Deployment View, graphical elements are either architectural elements (Nodes and Partitions) or instantiation of predefined components (Processors, Devices and Buses) that are declared in the TASTE Hardware Library.

Addition of a new graphical element to the current hardware model can be done through:

- The New menu of the main interface (picture on the left)
- The contextual menu in the representation of the model graphic (picture on the right)
- The toolbar of the main interface (picture below)
In the models browser tree, the Hardware Library can be found as a child of the DeploymentView node. The TASTE Hardware Library is named DV_Lib_Root and is composed of three sub nodes defining three categories of components that are available:

- Processors, representing computing subsystems (CPU, OS, memory...)
- Devices, representing bus communication interfaces
- Buses, representing hardware buses and supported communication protocols

To instantiate a component from the Hardware Library onto the current model, drag and drop it onto the diagram.

### 6.1.1.6.3 TASTE NODES

#### 6.1.1.6.3.1 Adding a TASTE Node

To add a TASTE Node to the model, use one of the three possibilities described in the chapter "6.1.1.6.2 Adding a new graphical element". Once selected, the TASTE editor enters a TASTE Node creation mode.

While in TASTE Node creation mode, the first click on the Deployment View diagram sets the upper left corner of the TASTE Node and the second one its lower right corner.

According to the TASTE modelling rules, the creation of a TASTE Node is legal if the Node fulfils a set of creation rules. These rules are checked on the fly by the TASTE graphical editor, depending on the position of the mouse. If the rules are fulfilled, the mouse pointer becomes a cross icon otherwise it becomes a wrong-way icon. The corresponding message is displayed in the status bar of the editor.

#### 6.1.1.6.3.2 Removing a TASTE Node

To remove a TASTE Node from the model, select the Node to remove then:

- Press the delete key on the keyboard
- In the “Edit” entry of the main menu, select the “Delete” entry
- In the toolbar, click on the “Delete” button

A TASTE Node can also be removed through the contextual menu of the Node, selecting the “Delete” entry.

#### 6.1.1.6.3.3 Editing the properties of a TASTE Node

To edit the properties of a TASTE Node, select the Node to edit then:

- In the “Edit” entry of the main menu select the “Edit Properties” entry
- In the toolbar, click on the “edit properties” button

Properties can also be edited either by double clicking on the Node to edit or by selecting the "Edit Properties" entry in the contextual menu of the Node to edit.

In edition mode, a popup dialog box named "Edit Data" is raised (image below). The dialog box is composed of the following two tabs:

- Node Attributes, to edit the direct attributes of the TASTE Node
Description, to set the description of the current TASTE Node in plain text format

**Node Attributes:**

- "Label" is the name of the current Node. When editing it, if the naming rules of the TASTE Node are not fulfilled, the string is displayed in red and the OK button remains inactive. The label can also be edited directly from the Deployment View diagram by clicking on it.
- "Fill Color" can be used to change the colour of the current Node

### 6.1.1.6.4. TASTE PROCESSORS

#### 6.1.1.6.4.1 Adding a TASTE Processor

While instantiating a TASTE Processor from the HW Library, automatic creations rules are applied:

- If the TASTE Processor is not created within an existing Node, a default Node is created
- A default Partition is created inside the Processor

While instantiating a TASTE Processor the drop action on the Deployment View diagram sets the upper left corner of either the TASTE Processor or the containing TASTE Node and the next click its lower right corner. The TASTE Processor is added into the created Node if needed. A default partition is added to the Processor.

According to the TASTE modelling rules, the creation of a TASTE Processor is legal if the Processor fulfils a set of creation rules. These rules are checked on the fly by the TASTE graphical editor, depending on the position of the mouse. If the rules are fulfilled, the mouse pointer becomes a cross icon otherwise it becomes a wrong-way icon. The corresponding message is displayed in the status bar of the editor.

#### 6.1.1.6.4.2 Removing a TASTE Processor

To remove a TASTE Processor from the model, select the Processor to remove then:

- Press the delete key on the keyboard
- In the “Edit” entry of the main menu, select the “Delete” entry
- In the toolbar, click on the "Delete" button

A TASTE Processor can also be removed through the contextual menu of the Processor, selecting the "Delete" entry.

#### 6.1.1.6.4.3 Editing the properties of a TASTE Processor

To edit the properties of a TASTE Processor, select the Processor to edit then:

- In the “Edit” entry of the main menu select the "Edit Properties" entry
- In the toolbar, click on the “edit properties” button
Properties can also be edited either by double clicking on the Processor to edit or by selecting the "Edit Properties" item in the contextual menu of the Processor to edit.

In edition mode, a popup dialog box named "Edit Data" is raised (image below). The dialog box is composed of the following three tabs:

1. Processor Attributes, to edit the direct attributes of the TASTE Processor
2. Report, to display the error/warning linked to the properties edition
3. Description, to set the description of the current TASTE Function in plain text format

**Processor Attributes:**

- "Label" is the instance name of the current Processor. While editing it, if the naming rules of the TASTE Processor are not fulfilled, the string is displayed in red and the OK button remains inactive. The label can also be edited directly from the Deployment View diagram by clicking on it.
- "Classifier" is a read only field that shows the classifier name of the current Processor, as it is defined in the HW Library.
- "Scheduling Protocol" is a read only field that shows the scheduling protocol of the current Processor, as it is defined in the HW Library. Note that TASTE uses "Posix_1003_Highest_Priority_First_Protocol".
- "Fill Color" can be used to change the colour of the current Processor

**6.1.1.6.5. TASTE PARTITIONS**

**6.1.1.6.5.1 Adding a TASTE Partition**

To add a TASTE Partition to the model, use one of the three possibilities described in the chapter "6.1.1.6.2 Adding a new graphical element". Once selected, the TASTE editor enters a TASTE Partition creation mode.

While in TASTE Partition creation mode, the first click on the Deployment View diagram sets the upper left corner of the TASTE Partition and the second one its lower right corner.

According to the TASTE modelling rules, the creation of a TASTE Partition is legal if the Partition fulfils a set of creation rules. These rules are checked on the fly by the TASTE graphical editor, depending on the position of the mouse. If the rules are fulfilled, the mouse pointer becomes a cross icon otherwise it becomes a wrong-way icon. The corresponding message is displayed in the status bar of the editor.

**6.1.1.6.5.2 Removing a TASTE Partition**

To remove a TASTE Partition from the model, select the Partition to remove then:

- Press the delete key on the keyboard
- In the "Edit" entry of the main menu, select the "Delete" entry
- In the toolbar, click on the "Delete" button

A TASTE Partition can also be removed through the contextual menu of the Partition, selecting the "Delete" entry.
6.1.1.6.5.3 Editing the properties of a TASTE Partition

To edit the properties of a TASTE Partition, select the Partition to edit then:

- In the Edit menu select the "Edit Properties" entry
- In the toolbar, click on the edit properties button

Properties can also be edited either by double clicking on the Partition to edit or by selecting the "Edit Properties" entry in the contextual menu of the partition to edit.

In edition mode, a popup dialog box named "Edit Data" is raised (image below). The dialog box is composed of the following two tabs:

- Partition Attributes, to edit the direct attributes of the TASTE Partition
- Description, to set the description of the current TASTE Partition in plain text format

**Partition Attributes:**

- "Label" is the name of the current Partition. When editing it, if the naming rules of the TASTE Partition are not fulfilled, the string is displayed in red and the OK button remains inactive. The label can also be edited directly from the Deployment View diagram by clicking on it.
- "Fill Color" can be used to change the colour of the current Partition
- "Coverage Enabled" is a Boolean flag that is used during system build process

6.1.1.6.6. TASTE DEVICES

6.1.1.6.6.1 Adding a TASTE Device

While instantiating TASTE Device from the HW Library, the drop action on the Deployment View diagram sets the upper left corner of the TASTE Device and the next click its lower right corner.

According to the TASTE modelling rules, the creation of a TASTE Device is legal if the Device fulfils a set of creation rules. These rules are checked on the fly by the TASTE graphical editor, depending on the position of the mouse. If the rules are fulfilled, the mouse pointer becomes a cross icon otherwise it becomes a wrong-way icon. The corresponding message is displayed in the status bar of the editor.

6.1.1.6.6.2 Removing a TASTE Device

To remove a TASTE Device from the model, select the Device to remove then:

- Press the delete key on the keyboard
- In the "Edit" entry of the main menu, select the "Delete" entry
- In the toolbar, click on the "Delete" button

A TASTE Device can also be removed through the contextual menu of the Device, selecting the "Delete" entry.
6.1.6.6.3 Editing the properties of a TASTE Device

To edit the properties of a TASTE Device, select the Device to edit then:

- In the Edit menu select the "Edit Properties" entry
- In the toolbar, click on the "edit properties" button

Properties can also be edited either by double clicking on the Device to edit or by selecting the "Edit Properties" entry in the contextual menu of the Device to edit.

In edition mode, a popup dialog box named "Edit Data" is raised (image beside). The dialog box is composed of the following three tabs:

- "Driver Attributes", to edit the direct attributes of the TASTE Device
- "Help", a text field to provide additional information to the build process
- "Description", to set the description of the current TASTE Device in plain text format

**Driver Attributes:**

- "Label" is the instance name of the current Device. While editing it, if the naming rules of the TASTE Device are not fulfilled, the string is displayed in red and the OK button remains inactive. The label can also be edited directly from the Deployment View diagram by clicking on it.
- "Classifier" is a read only field to show the classifier name of the current Device. Its value is defined in the HW Library.
- "Config" is a field used to set the configuration of the current Device
- "ASN.1 Type" is a button to popped up a read only editor to display the ASN.1 definition of the configuration type of the current Device
- "Id and version" is a read only field to display the version of the current Device. Its value is defined in the HW Library.
- "Fill Color" can be used to change the colour of the current Device
- "Comment" is a free text field to add a comment to the current Device

6.1.6.7. TASTE BUSES

6.1.6.7.1 Adding a TASTE Bus

While instantiating a TASTE Bus from the HW Library, the drop action on the Deployment View diagram sets the upper left corner of the TASTE Bus and the next click its lower right corner.

According to the TASTE modelling rules, the creation of a TASTE Bus is legal if the Device fulfils a set of creation rules. These rules are checked on the fly by the TASTE graphical editor, depending on the position of the mouse. If the rules are fulfilled, the mouse pointer becomes a cross icon otherwise it becomes a wrong-way icon. The corresponding message is displayed in the status bar of the editor.

6.1.6.7.2 Removing a TASTE Bus

To remove a TASTE Bus from the model, select the Bus to remove then:

- Press the delete key on the keyboard
6.1.1.6.7.3 Editing the properties of a TASTE Bus

To edit the properties of a TASTE Bus, select the Bus to edit then:

- In the “Edit” menu select the “Edit Properties” entry
- In the toolbar, click on the “edit properties” button

Properties can also be edited either by double clicking on the Device to edit or by selecting the “Edit Properties” entry in the contextual menu of the Bus to edit.

In edition mode, a popup dialog box named "Edit Data" is raised (image beside). The dialog box is composed of the following three tabs:

- "Bus Attributes", to edit the direct attributes of the TASTE Bus
- "Description", to set the description of the current TASTE Bus in plain text format

**Bus Attributes:**

- "Label" is the instance name of the current Bus. While editing it, if the naming rules of the TASTE Bus are not fulfilled, the string is displayed in red and the OK button remains inactive. The label can also be edited directly from the Deployment View diagram by clicking on it.
- "Classifier" is a read only field to display the classifier name of the current Bus. This name is defined in the HW Library.
- "Fill Color" can be used to change the colour of the current Bus
- "Comment" is a field to add a comment to the current Bus

6.1.1.6.8. TASTE DV CONNECTIONS

In the Deployment View, connections can be drawn only between a TASTE Device and a TASTE Bus (or the reverse way). The connection is created between interfaces provided by the TASTE components. An interface is represented as a black square on the bounding box of the component (see picture beside).

A TASTE Device provides one and only one interface to be connected to a BUS.

At creation time, a TASTE Bus provides two interfaces to connect Devices. However, additional interfaces may be added later on if needed.

To add a new interface to a Bus, either open the contextual menu of the Bus and select the "New Interface" entry, or select the "New Interface" button in the toolbar.
To connect two interfaces either drag and drop from the source Interface to the target Interface, or right click on the source Interface, select the "Connect" entry from the contextual menu then click on the target Interface.

### 6.1.1.6.9. SW-HW BINDINGS

The binding phase consists in deploying the software model designed in the Interface View into the current hardware model described by the Deployment View.

Bindable software elements in an IV model are:
- Terminal TASTE Functions
- TASTE IV Connections between two terminal TASTE Functions that are bound to two different Nodes.

Once bound, these elements are displayed in blue in the models browser.

#### 6.1.1.6.9.1 Binding of TASTE Functions

To bind all the TASTE Functions in one single partition, open the contextual menu of the chosen partition then select the "Bind All" entry (see picture on the right). This option is available only if none of the TASTE Functions from the software model is bound.

To bind a single TASTE Function to a Partition, either open the contextual menu of the chosen partition and select the Function to bind in the provided list under the "Bind" entry (see picture on the left) or drag and drop the Function from the models browser to the chosen Partition.

According to the TASTE modelling rules, the binding of a TASTE Function is legal if the Function fulfils a set of binding rules. These rules are checked on the fly by the TASTE graphical editor. While binding using drag and drop, verification relies on the position of the mouse. If the rules are fulfilled, the mouse pointer becomes a cross icon otherwise it becomes a wrong-way icon. The corresponding message is displayed in the status bar of the editor. While binding using the contextual menu, only authorized items are proposed in the Bind list.

As a shortcut, the designer is allowed to drag and drop a non terminal TASTE Function into a Partition. That will result in the binding of all the contained terminal Functions to the chosen Partition.

#### 6.1.1.6.9.2 Unbinding of TASTE Functions

To unbind all the TASTE Functions bound to one Partition, open the contextual menu of the Partition and select the "Unbind All" entry (see picture on the right).

To unbind a single TASTE Function, either open the contextual menu of the chosen Partition and select the Function to unbind in the provided list under the "Unbind" entry (see picture on the left) or open the contextual menu of the models browser and select the "Unbind" entry (see picture on the right).
6.1.6.9.3 Binding of connections from the Interface View

To bind all the TASTE connections to one single Bus, open the contextual menu of the chosen Bus then select the "Bind All" entry (see picture on the right). This item is available only if none of the TASTE connections is bound.

To bind a single TASTE connection to a Bus, either open the contextual menu of the chosen Bus and select the connection to bind in the provided list under the "Bind" entry (see picture on the left) or drag and drop the connection from the models browser to the chosen Bus.

According to the TASTE modelling rules, the binding of a TASTE connection is legal if the connection fulfills a set of binding rules. These rules are checked on the fly by the TASTE graphical editor. While binding using drag and drop, the verification relies on the position of the mouse. If the rules are fulfilled, the mouse pointer becomes a cross icon otherwise it becomes a wrong-way icon. The corresponding message is displayed in the status bar of the editor. While binding using the contextual menu, only authorized items are proposed in the Bind list.

6.1.6.9.4 Unbinding of connections from the Interface View

To unbind all the TASTE connections bound to a Bus, open the contextual menu of the Bus and select the "Unbind All" entry (see picture on the right).

To unbind a single TASTE connection, either open the contextual menu of the chosen Bus and select the connection to unbind in the provided list under the "Unbind" entry (see picture on the left) or open the contextual menu of the connection to unbind in the models browser and select the "Unbind" entry (see picture on the right).

6.1.1.7. TIMING ANALYSIS (CONCURRENCY VIEW EDITOR)

After the Deployment View model has been completed, it must be automatically transformed to elaborate an intermediate representation before being able to generate the target source code. This intermediate representation is another AADL model that includes explicit real-time constructs while preserving the architecture and non functional properties specified in the IV and DV models. It is called the Concurrency View (CV in short) and should never be edited directly.

From within the TASTE editor, the CV model is automatically generated when the user select the "Concurrency View" tab. In addition to guiding code generation, the CV model can also be used to perform early model driven timing analysis and adjust a few real-time attributes within the Concurrency View Editor. The timing analysis tools that are embedded in the TASTE CV editor are:

- Cheddar, a scheduling analysis tool developed by the University of Brest (http://beru.univ-brest.fr/~singhoff/cheddar/)
- Marzhin, an AADL run time simulator developed by Ellidiss Technologies and Virtualys.

In order to get theoretical timing analysis results from the Cheddar tool, the user has to click on the "Theoretical schedulability test" button. Static simulation and timing analysis results from this simulation can be computed by clicking, respectively, on the "Cheddar schedule table" and "Simulation schedulability test" buttons.
The Marzhin tool offers dynamic simulation of the CV model. The simulation can be controlled with the "play" and "stop" button.

Note that the Concurrency View editor has only been used for the ERGO project to modify stack sizes for each thread generated in the applications.

6.1.2. TASTE SW BUILDING

The build process consists in automatically generating the executable files for the set of IV Functions that are bound to each DV Partition. This is performed by several tools that are part of the TASTE tool-chain, especially Build Support and Ocarina.

6.1.2.1. PREREQUISITES

To start the build process, the current edited model shall fit the following requirements:

- DataView, InterfaceView and DeploymentView shall be properly loaded. If the TASTE scripts are used (taste-create-project and taste-edit-project), the three files are all loaded at start time. An error is raised if requirement not fulfilled.
- Each terminal function of the Interface View shall be bound to a Partition in the Deployment View. Otherwise an error is raised.
- Each connection between two terminal Functions which are bound to two different Partitions shall be bound to a Bus. Otherwise an error is raised.
- For each sporadic or cyclic provided interface in the InterfaceView, an execution thread will be created.

6.1.2.2. BUILD PROCESS

From the graphical editor of the TASTE tool chain, the build process is started using the "Tools" menu and selecting "Build the system (in C)".

One executable per partition defined in the Deployment View is generated in the directory [root_model_dir]/binary.c/binaries. Their names are the same as the name of the corresponding Partition defined in the Deployment View.

6.1.2.3. BUILD TAILORING

6.1.2.3.1. BUILDING MORE THAN ONE EXECUTABLE ON A NODE

To generate two executables running on the same processor, the user must create two identical TASTE Nodes in the Deployment View and connect them through an IP bus. In the configuration property of the two Devices, the IP address must be the identical. During the code generation process, the TASTE tool chain will recognize this pattern and generate the code and makefiles to generate the two executables.

6.1.2.3.2. MODIFYING EXECUTION THREADS

To configure each execution thread TASTE provides the possibility of modifying some properties as the stack size or its priority. ConcurrencyView model controls the configuration of real time.

In the tab of ConcurrencyView it is possible to click the button "Edit Properties" and modify the following real time properties of each thread:

- A **priority** level
- **Stack size** of memory
- **Phase** time
6.1.2.3.3. TASTE DIRECTIVES

TASTE directives are used to add command line options either to the compiler or to the linker during the build process.

TASTE directives are defined at TASTE function level during interface view modelling. They are modelled as context parameters of type "TASTE_Directive". Their value can be:

- **compiler-option**: "[command line option to add]", to add a command line option to the compiler for the current TASTE function
- **linker-option**: "[command line option to add]", to add a command line option to the linker for the current TASTE function

6.1.2.3.4. PREDEFINED SCRIPTS

The designer can tailor the build process using the three predefined files: "user_init_pre.sh", "user_init_post.sh" and "user_init_last.sh". When used, these files shall lay in the main directory of the project. This directory contains the "interfaceview.aadl" and "deploymentview.aadl" files.

The three predefined scripts are bash scripts. There are no restrictions for the designer in the usage of bash commands in the definition of the scripts.

If the "user_init_pre.sh" file exists, the script is executed before the initialisation of the TASTE build process.

If the "user_init_post.sh" file exists, the script is executed at the end of the initialisation of the TASTE build process.

If the "user_init_last.sh" file exists, the script is executed at the end of the TASTE build process.

6.2. V&V APPROACH & BIT TOOLS

6.2.1. V&V APPROACH

The V&V approach with the BIP tools is illustrated in Figure 6-3. This process starts with a system modelled in BIP consisting of the nominal model (how the system behaves when all environment assumptions are met) and possibly of the faulty model (how the system behaves in case of errors). The BIP model can be obtained from a TASTE design by a model translation implemented in TASTE2BIP (developed in OG1). TASTE2BIP expects as input an interface view modelling the software architecture with components implemented in SDL and communicating via signals, and a data view describing user data types. The TASTE model usually describes the nominal behaviour of the system and possibly defines an FDIR (fault detection, isolation and recovery) component. The faulty model can be described directly in the BIP model.

The nominal BIP model can be subject to invariant-based compositional analysis with the iFinder/iChecker tool. iFinder compositionally computes the system invariants based on the options given, where an invariant is a logical formula abstracting the set of reachable states of the system. iChecker checks the satisfaction of a safety property on a BIP model by using the computed invariants. For this it interacts with the Z3 SMT Solver that provides a yes/no verdict and a counterexample in case the property is not satisfied. The property cannot be satisfied in two cases: either the invariants are not sufficient and additional invariants needs to be added by the user for the requirement to be satisfied, or the property does not hold on the model. Therefore the counterexample needs to be checked by the user and appropriate actions should be taken.

A more thorough validation is done with SMC-BIP implementing statistical model-checking techniques. A qualitative or quantitative property is specified in PBLTL/MTL which is checked on the several system simulations. A yes/no verdict is given for the satisfaction of a qualitative property, while a satisfaction probability is computed in the case of a quantitative property. The tool is developed in OG1.
The TASTE/BIP model can define components with FDIR functionalities. The FDIR tool implements
synthesis algorithms to obtain FDIR components from the system design, property to enforce and
recovery strategies. The synthesized FDIR component consists of a diagnoser which detects based
on the observed events whether a fault has occurred, and a controller which is triggered in case of
a fault to bring the system back in a safe state. An FDIR implementation in C++ is generated for
this component, as mentioned below. Please note that this tool is based on a methodology using
formal methods that targets event-based timed properties, e.g., whether some event $x$ happens
before a timeout $t$. Data-based safety properties, e.g., if the value $v$ of an event parameter is below
a threshold $\text{min}$ then the sender component needs to be stopped, are not handled by this tool.

Finally, the BIP Compiler and Engine (under development in OG1) allows generating C++ code from
a BIP model and executing it. This tool is the reference for the FDIR tool described above. The BIP
compiler has been used in the orbital and planetary scenarios to generate the C++ code for the
FDIR components. The BIP real-time engine has been used to integrate and execute the BIP
generated FDIR component into the starting TASTE design.

![Figure 6-3. V&V approach with the BIP Tools](image)

### 6.2.2. FDIR TOOL

The FDIR tool takes as input a BIP model. The BIP model could consist of two packages: one
package contains the nominal behaviour of the system and one package contains the faulty
behaviour. Then the extended model is obtained by calling the *merging* filter as shown in Figure
6-4. The FDIR analysis is done on the extended model. The reachability graph is built with respect
to observable events and faults are checked for being diagnosable. The tool stops if there is a
non-diagnosable fault. If the faults are diagnosable, the diagnoser part of the FDIR is synthesized. The
controller part is to be given by the user in BIP format. All components are put into one BIP model
on which the properties to ensure are validated.

Please note that the tool works only for faults detectable by events. If the property to ensure is
based on data, as it is the case of the planetary and orbital scenarios, the FDIR component is
modelled by the user in BIP and goes through the validation process with a faulty environment.

Once the FDIR component is obtained and validated, C++ code is generated and the component
can be integrated in the original TASTE design as detailed in Section 11.2.3.
6.2.3. IFINDER

We show in the following how to use the iFinder/iChecker tool on abstract models of the orbital and planetary scenarios.

6.2.3.1. ORBITAL SCENARIO USE CASE

The abstract model of the orbital scenario focuses on the interaction between the Agent and the RARM_Controller. This model has been manually constructed such that to not use any data variables. That is, all relevant variables have been "pushed" towards the control (locations expanded) and/or structure (interface expansion), while the others have been fully ignored. The resulting abstract model contains only components with pure timed automata behaviour (no data, only real-time clocks), interconnected by primitive calls (with no data transfer).

The abstract model has been developed manually with TASTE and SDL and then translated automatically by TASTE2BIP. In BIP, the abstract model contains 12 atomic components interconnected by 28 connectors.

Generation of Invariants

Most of the invariant generation techniques provided by iFinder are running successfully on the abstract model. We summarize below how to run the tool and the most interesting results obtained.

- linear - this analysis computes a specific category of place-transition invariants for BIP components, atomic or composite. The analysis completes successfully in 1.5 seconds and generates 22 independent linear constraints.

```
$ ifinder.sh -ct Syst -a linear -p OrbitalAbstractModel -v
[ifinder] load package 'OrbitalAbstractModel' ... ok
[ifinder] locate compound type 'Syst' ... ok
[ifinder] instantiate component type 'Syst' ... ok
[ifinder] check if 'linear' is applicable ... yes
[ifinder] generate invariant by 'linear' ...
[ifinder] construct the Petri net ... |P|=170, |T|=281 done
[ifinder] extract 22 linear constraints
[ifinder] generate invariant by 'linear' done
[ifinder] done
1.436u 0.056s 0:00.71 208.4% 0+0k 0+32io 0pf+0w
```
zone-reachability - this analysis computes the set of the reachable symbolic states (states + clock constraints) of a BIP component, atomic or compound. The analysis completes successfully in 13.5 seconds, and generates a state graph with 17571 symbolic states.

```
$ ifinder.sh -ct Syst -a zone-reachability -p OrbitalAbstractModel -v
[ifinder] load package 'OrbitalAbstractModel' ... ok
[ifinder] locate compound type 'Syst' ... ok
[ifinder] instantiate component type 'Syst' ... ok
[ifinder] check if 'zone-reachability' is applicable ... yes
[ifinder] generate invariant by 'zone-reachability' ...
[ifinder] construct the Petri net ... |P|=170, |T|=281, |C|=10 done
[ifinder] reached 17571 symbolic states
[ifinder] generate invariant by 'zone-reachability' done
[ifinder] done
13.540u 0.400s 0:10.04 138.8%   0+0k 0+23944io 0pf+0w
```

trap - this analysis extracts a specific category of place-transition invariants for BIP components. Its execution can be limited by the maximal number of trap invariants to be generated. For example, the analysis completes successfully in 12.5 seconds the extraction of 200 trap invariants.

```
ifinder.sh -ct Syst -a trap -p OrbitalAbstractModel -xtl 2000 -v
[ifinder] load package 'OrbitalAbstractModel' ... ok
[ifinder] locate compound type 'Syst' ... ok
[ifinder] instantiate component type 'Syst' ... ok
[ifinder] check if 'trap' is applicable ... yes
[ifinder] generate invariant by 'trap' ...
[ifinder] construct the Petri net ... |P|=170, |T|=281 done
[ifinder] extract 2000 traps
[ifinder] generate invariant by 'trap' done
[ifinder] done
12.472u 0.464s 0:10.42 124.0%   0+0k 0+7592io 0pf+0w
```

Moreover, the analysis completes the extraction of the entire set of traps. In this case, it generates 229621 trap invariants.

Finally, we shall mention that some other analysis in ifinder have failed to complete due to state explosion problems:

- control-reachability - this analysis constructs the reachable control states for a BIP component, atomic or compound, regardless of any data and timing constraints. This analysis failed on the model; however, this is not surprising as the overall behaviour is significantly driven by the time (through the activation of periodic interfaces). When timing constraints are ignored, the overall behaviour becomes highly non-deterministic and all combinations of states are potentially reachable.

- zone-reachability with history clocks - this analysis constructs the reachable control states for BIP components, atomic or composite, extended with history clocks for tracking interaction time on exported ports (interface). This analysis fails both for Agent and RARM_Controller as their interface is relatively complex (15 and 13 exported ports respectively), as well as their behaviour is not much constrained by time. Nonetheless, the analysis may succeed if a lower number of ports are considered observable.

Verification of Safety Properties

The invariants generated previously were used to prove some interesting safety properties on the abstract model. We summarize below some results:

- Agent never failed - the 'Failed' state of the Agent is entered whenever the component receives an unexpected message / interface call in some specific state (e.g., while waiting for a pick execution a dropping status is received). This property is successfully verified using the zone-reachability invariant, in about 23 seconds. Notice moreover that the verification fails using other (weaker) invariants.

```
$ ichecker.sh -p OrbitalAbstractModel -r Syst -i reachable.inv -s agent_not_failed.pro -v
```
RARM Controller never failed - the 'Failed' state of the RARM_controller is entered whenever the component receives an unexpected message / interface call in some specific state (e.g., a drop request is received before a pick request). This property is successfully validated using the zone reachability invariant in about 23 seconds. Notice moreover that the verification fails using other (weaker) invariants.

```bash
$ ichecker.sh -p OrbitalAbstractModel -r Syst -i reachable.inv -s rarm_not_failed.pro -v
```
6.2.3.2. PLANETARY SCENARIO USE CASE

Experiments have been conducted on the complete planetary scenario with focus on the interaction between the Agent and the RARM_Controller. The model has been developed with TASTE and SDL and then translated automatically with TASTE2BIP. This model is the one used for the design of the FDIR component and its validation by simulation. In BIP, the model contains 40 atomic components interconnected by 88 connectors. Furthermore, it uses external code for computations on variables of ASN.1 types.

Generation of Invariants

Some of the invariant generation techniques were run successfully on our planetary model. We report hereafter the results obtained by linear and trap invariant computation methods.

- **linear** - this analysis computes a specific category of place-transition invariants for BIP components, atomic or composite. The analysis completes successfully in 2.5 seconds and generates 71 independent linear constraints.

```bash
$ ifinder.sh -p ModelFDIRfull -ct Syst -a linear -v
[ifinder] load package 'ModelFDIRfull' ... ok
[ifinder] locate compound type 'Syst' ... ok
[ifinder] instantiate component type 'Syst' ... ok
[ifinder] check if 'linear' is applicable ... yes
[ifinder] generate invariant by 'linear' ...
[ifinder] construct the Petri net ... |P|=634, |T|=887 done
[ifinder] extract 71 linear constraints
[ifinder] generate invariant by 'linear' done
[ifinder] done
2.360u 0.088s 0:01.69 144.3%    0+0k 0+96io 0pf+0w
```

- **trap** - this analysis extracts a specific category of place-transition invariants for BIP components. For example, the analysis completes successfully in 5.5 seconds for the extraction of 867 trap invariants.

```bash
$ ifinder.sh -p ModelFDIRfull -ct Syst -a trap -v
[ifinder] load package 'ModelFDIRfull' ... ok
[ifinder] locate compound type 'Syst' ... ok
[ifinder] instantiate component type 'Syst' ... ok
[ifinder] check if 'trap' is applicable ... yes
[ifinder] generate invariant by 'trap' ...
[ifinder] construct the Petri net ... |P|=634, |T|=887 done
[ifinder] extract 867 traps
[ifinder] generate invariant by 'trap' done
[ifinder] done
5.476u 0.464s 0:05.05 117.4%    0+0k 0+2376io 0pf+0w
```

Finally, we shall mention that additional methods were not applicable for different reasons. Zone reachability is restricted to component models in form of regular timed automata, with no data, and henceforth does not run on almost any of the atomic components, neither on the composition. Control-reachability runs on atomic components, however, it fails on the composition due to state explosion.
Verification of Safety Properties

As the invariants generated for the entire model are rather weak, we were able to prove only specific properties related to the state of components and their specific near interfaces. These local properties are naturally derived from the linear and trap invariants.

For example, we can prove the following mutual exclusion property relating the Antenna component and Antenna_ANTENNA_REQUEST interface, that is, always

\[
\text{Antenna.Request + Antenna.Communicating + Antenna.l[2, 4, 5, 6, 8, 9, 10] + Antenna\_ANTENNA\_REQUEST.l1 = 1}
\]

The property is proven in about 6.5 seconds using the linear and trap invariants.

```
$ ichecker.sh -i inv_1.inv -s antenna.pro -p ModelFDIRfull -r Syst -v
[ichecker] load package 'ModelFDIRfull' ... ok
[ichecker] locate component type 'Syst' ... ok
[ichecker] instantiate component type 'Syst' ... ok
[ichecker] process invariant specification 'inv_1.inv' ...
[ichecker] process line '-ct Syst -a trap' ...
[ichecker] locate compound type 'Syst' ... ok
[ichecker] instantiate component type 'Syst' ... ok
[ichecker] check if 'trap' is applicable ... yes
[ichecker] generate invariant by 'trap' ...
[ichecker] construct the Petri net ... |P|=634, |T|=887 done
[ichecker] extract 867 traps
[ichecker] generate invariant by 'trap' done
[ichecker] invariant recorded for ''
[ichecker] process line '-ct Syst -a linear' ...
[ichecker] locate compound type 'Syst' ... ok
[ichecker] instantiate component type 'Syst' ... ok
[ichecker] check if 'linear' is applicable ... yes
[ichecker] generate invariant by 'linear' ...
[ichecker] construct the Petri net ... |P|=634, |T|=887 done
[ichecker] extract 71 linear constraints
[ichecker] generate invariant by 'linear' done
[ichecker] invariant recorded for ''
[ichecker] process invariant specification 'inv_1.inv' done
[ichecker] load property 'antenna.pro' ... done
[ichecker] generate SMT script 'inv_1-antenna' ... done
[ichecker] run SMT solver ... valid
6.440u 0.512s 0:05.24 132.6% 0+0k 0+2544io 0pf+0w
```
7. PLANNING MODELLING (PDDL MODELS)

The Planning Domain Definition Language (PDDL) was created in order to define a standard planning language and to allow comparative analysis of the different planning systems. PDDL was also developed as the planning input language of the First International Planning Competition (IPC), whose objective was to compare the state-of-the-art planning systems. The PDDL version used in this Project is 2.2 with an extension for external functions. This version supports:

- Durative Actions which are PDDL actions with a duration, some derived from a linear equation. For example, the duration of the action going to depends, among other parameters, on the distance to be traversed.

- Plan metrics to evaluate the plan quality and maximize or minimize the cost.

- Numeric fluents which represents reusable and consumable resources. In both cases an event captures the concept of resource usage, but with one difference: for reusable resources an event represents an amount of a given resource booked on a temporal interval while for a consumable resource an event is an amount produced or consumed in a time instant.

- Timed Initial Literals which are used to represent uncontrollable events that will happen in a known future time. For example, to represent communication time windows.

- External functions are independent component, which provide information that can be used by the mission planner during search. These functions are defined as numerical fluents in a specific point in the domain file.

::modules
  (:module path-finding
    (:function (get_energy_needed ?from - waypoint ?to - waypoint))
    (:function (get_time_needed ?from - waypoint ?to - waypoint))
  )
  (:module battery
    (:function (energy_available ?r - rover))
  )
::durative-action move
  (:parameters (?r - rover ?from - waypoint ?to - waypoint)
  (:duration (= ?duration (get_time_needed ?from ?to))
  (:conditions (and
      (at-start (at ?r ?from))
      (at-start (>= (energy ?r) (get_energy_needed ?from ?to)))
  (:effects (and
      (at-start (not (at ?r ?from)))
      (at-end (at ?to)))
      (at-end (= (energy ?r) (get_energy_needed ?from ?to)))
  ))

This example shown the definition of two external functions. External functions are defined as modules in the PDDL domains. The first function is called module_name_1 and provide 2 different functions and the second function is called module_name_2 and provide 1 function. The syntax of each functions is like a PDDL predicate and the name of the module must be the same at the library, because when during the loading process the name of the library is built using the name of the module at the same way.
The previous actions uses the external functions defined before. The duration of the actions is computed using the external function get_time_duration and the energy needed and used is getting by the external function get_energy_needed.

This PDDL version does not support: (1) Non-Durative Actions; (2) ADL extensions (i.e. disjunctive preconditions, conditional effects, forall/exists quantifiers); (3) Numeric effects referring to duration of the durative actions and (4) Continuous numeric effects during the execution of an action.

7.1. REAL VALUES TO PDDL OBJECTS

Useful values used in ERGO are likely to include integers, floats and doubles. These are used to provide inputs to the planner, and are placed within the problem file for instances of planning. However, the PDDL standard is restrictive in the values allowed to be defined in the problem. The Stellar planner is not adherent to all of these, but does require that all object instances generated from the values do not begin with a non-alphabet character.

It is therefore required that such values are never sent to the planner, otherwise it will fail. As incoming values from goals should not be restricted in this way, the reactor automatically prepends an ‘n’ character to the beginning of each object created from goal values dynamically sent to the reactor. However, as the domain and problems share the constant objects defined in the domain, the constants in the domain should be defined with this prefix at initialisation. The problem will then be modified to match within the reactor, and further state updates and new goal parameters will be made to match this convention at execution time.
8. AGENT DEVELOPMENT

8.1. REACTORS INSTANTIATION

The second part of the instantiation will consist of the development of the reactors that conform the agent and to prepare the agent itself. It is assumed that the already existing reactors in ERGO (in particular GCI, Command Dispatcher, and Mission Planner) will be reusable, making the corresponding configuration files. In the following, we will describe how to particularize the Ground Control Interface and the Command Dispatcher Reactor, and the process to add a new reactor.

8.1.1. ADDING A NEW REACTOR

One of the most important components of ERGO is the agent main controller, based on T-REX. The controller follows the "Sense-Plan-Act" paradigm. [RD.31] Time is discretized, and it is responsibility of the controller or agent to control a set of coordinated control loops, also known as "reactors" to perform the execution. The controller follows an algorithm that is described in [RD.31].

To add a new functionality to the system, a new reactor can be easily added. To add a new reactor, a few steps can be followed to ease the process:

- Define the nature of the reactor.
- Define the timelines needed.
- Code the reactor functionality. Can be done as an isolated library.
- Implement the reactor as an interface between the rest of the system and the previous defined functionality.

8.1.1.1. DEFINE THE NATURE OF THE REACTOR

Reactors can be reactive or deliberative. Reactive reactor will execute the goals immediately and don't need to deliberate. Deliberative reactors need time to deliberate, and may be busy for more than one tick.

8.1.1.2. DEFINE THE TIMELINES NEEDED

Define the timelines needed for the reactor to work and communicate with the system. Reactors can have internal timelines and external timelines.

There are a set of rules and guidelines regarding the timelines:

- A timeline is owned (internal) by one and only one reactor.
- Only the reactor that owns a timeline can post observations (facts) on it.
- All reactor that are subscribed (external) to a timeline will receive this observations and can post goals as requests for actions to the timelines.
- The reactor should be subscribed to all the timelines which state is needed and all the ones it may want to command.
- Internal timelines are used to publish the state of the reactor and to give an interface to accept goals form other reactors.

The timelines work like state machines, where only one predicate (state) can be true at any moment. The states are set by the observations published on the timeline. The definition of a new internal timeline starts by defining the state machine. Each of the states is a predicate, composed by parameters. All the timelines are defined in a timeline configuration file. A generic example of this file can be seen in Figure 8-1. A real example of the configuration file can be seen in Figure 8-2.
XML TIMELINE CONFIGURATION

```xml
<?xml version="1.0" encoding="utf-8"?>
<TimelineConfig>

<Timeline name="__timeline1__" tmname="T__Tmcode1__" rcname="R__Tmcode1__">
  <inactivePred name="__name1__" tcname="0"/>
  <errorPred name="__name2__" tcname="0"/>
</Timeline>

<Timeline name="__timeline2__" tmname="T__Tm_code2__" rcname="R__Tm_code2__">
  <inactivePred name="__name3__" tcname="0"/>
  <activePred name="__name4__" tcname="C__TCcode1__">
    <parameter name="__param_name1__" type="__type__" default="__option__"/>
    <parameter name="__param_name2__" type="__type__" default="__option__"/>
  </activePred>
  <errorPred name="__name4__" tcname="0"/>
</Timeline>

</TimelineConfig>
```

**Figure 8-1: XML generic timeline configuration**

The configurable parameters are:

- **Name __timeline1__ and __timeline2__**: The name of the timeline. This name should be in lowercase for a correct communication with the Planner Reactor.

- **Timeline code __Tmcode1__ and __Tmcode2__**: A code for the timeline. The code is in upper case and represents the timeline. It is used to make the telemetry and the recall code. The code with the prefixes T (for telemetry) and R (for recall) can't be the same as one of the reserved codes (TOBT, TICK, TAUL, TLTC, TTQN, TT, PNGS, PG, RQ, RP, OBCP, RALL, KILL). This code provides needed information to the Ground Control Interface. If there is no recall code, it can be set as "0".

- **The predicates can be active, inactive or error.**
  - **Active predicates**: Those are the predicates that are goleable, can receive goals.
  - **Inactive predicates**: These predicates can't receive goals. Just a status of the timeline.
  - **Error predicates**: These predicates represent an error state. These are also not goleables.

- **Name of the predicate __name1__, __name2__, etc**: name of the predicate inside the timeline. Each predicate in a timeline should have a different name. The name should be in lowercase for a correct communication with the Planner Reactor.

- **Tcname __TCcode1__**: A code that represents the telecommand to send E1 and E2 goals. This should be a short upper case code. It is used by the Ground Control Interface to parse the Telecommands received. The complete telecommand is preceded by a C. If the predicate does not have a telecommand, it is represented by "0". Inactive predicates and error predicates cannot have telecommands. It cannot be one of the reserved codes listed in the __Tmcode__ bullet.

- **Name of the parameter __param_name1__ and __param_name2__**: The name for each of the parameters of a predicate. It should be unique inside a predicate and in lower case for a correct communication with the planner reactor.

- **Type of the parameter __type__**: Data type of the parameter. There are different types of data available. Each have a different set of parameters. These available types are shown in Table 8-1.

- **Option of the parameter __option__**: An extra option for the parameter. This options are used in case a telecommand with a missing parameter is received. The possible options are:
- A value representing a default value. This value will be set if the corresponding value is missing in the telecommand.

- "needed". If a parameter set as needed, the telecommand will be discarded and an error shown in terminal.

- "lastObs". This parameter will be set to the equivalent in the last observation, or the goal will be created without it. It is set for the parameters that are not essential or can be deduced afterwards.

- With all this information, a timeline is defined and ready for its usage by all the generic reactor of ERGO. Inside the new reactor it is important to follow the names set in the configuration file.

<table>
<thead>
<tr>
<th>Data type name</th>
<th>arguments</th>
<th>Arguments Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>min</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>Integer</td>
</tr>
<tr>
<td>float</td>
<td>min</td>
<td>Float</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>Float</td>
</tr>
<tr>
<td>point2</td>
<td>X</td>
<td>Double</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Double</td>
</tr>
<tr>
<td>point3</td>
<td>X</td>
<td>Double</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Double</td>
</tr>
<tr>
<td></td>
<td>Z</td>
<td>Double</td>
</tr>
<tr>
<td>string</td>
<td>str</td>
<td>std::String</td>
</tr>
<tr>
<td>rectangle</td>
<td>Xmin</td>
<td>Double</td>
</tr>
<tr>
<td></td>
<td>Ymin</td>
<td>Double</td>
</tr>
<tr>
<td></td>
<td>Xmax</td>
<td>Double</td>
</tr>
<tr>
<td></td>
<td>Ymax</td>
<td>Double</td>
</tr>
</tbody>
</table>
XML TIMELINE CONFIGURATION EXAMPLE

```xml
<?xml version="1.0" encoding="utf-8"?>
<TimelineConfig>

<Timeline name="roboticarm" tmname="TORA" rcname="RORA">
  <inactivePred name="idleat" tcname="0">
    <parameter name="atslotid" type="slot" default="needed"/>
  </inactivePred>
  <inactivePred name="picked" tcname="0">
    <parameter name="apmid" type="apm" default="needed"/>
    <parameter name="atslotid" type="slot" default="lastObs"/>
  </inactivePred>
  <activePred name="moving" tcname="CMOVE">
    <parameter name="fromslotid" type="slot" default="lastObs"/>
    <parameter name="toslotid" type="slot" default="needed"/>
  </activePred>
  <activePred name="picking" tcname="CPICK">
    <parameter name="apmid" type="apm" default="needed"/>
    <parameter name="fromslotid" type="slot" default="lastObs"/>
    <parameter name="toslotid" type="slot" default="needed"/>
  </activePred>
  <activePred name="dropping" tcname="CDROP">
    <parameter name="apmid" type="apm" default="needed"/>
    <parameter name="fromslotid" type="slot" default="lastObs"/>
    <parameter name="toslotid" type="slot" default="needed"/>
  </activePred>
  <errorPred name="fault" tcname="0"/>
</Timeline>

<Timeline name="battery" tmname="TBAT" rcname="0">
  <inactivePred name="set" tcname="0">
    <parameter name="level" type="int" default="lastObs"/>
  </inactivePred>
</Timeline>

</TimelineConfig>
```

Figure 8-2: XML example Timeline configuration file

8.1.1.3. CODE THE REACTOR FUNCTIONALITY

The main functionality of the new reactor can be done as a complete separate library. This can ease the process of testing the functionality. All the reactor already available in ERGO are divided into functionality and reactor. For example, the GCI folder has two subfolders, gcifunc and gcireactor.

8.1.1.4. IMPLEMENT THE REACTOR

The teleoreactor class, the base class for all reactors, has a set of methods that interface with the agent (see Table 8-2), and a set of methods that have to be overloaded when defining a new reactor (see Table 8-3).
Table 8-2: Methods called by the agent for this reactor to be functional

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void newTick(Tick);</td>
<td>Called by the agent at the beginning of a new tick. Updates internal</td>
</tr>
<tr>
<td></td>
<td>representation of the tick and state of goals pending, dispatched or</td>
</tr>
<tr>
<td></td>
<td>active. Calls handleTickStart of available each reactor.</td>
</tr>
<tr>
<td>bool doSynchronize();</td>
<td>Called by the agent to check if the synchronization of the reactor was</td>
</tr>
<tr>
<td></td>
<td>successful, if not this method posts a failed observation. Calls</td>
</tr>
<tr>
<td></td>
<td>synchronize of each available reactor.</td>
</tr>
<tr>
<td>void doNotify(Observation);</td>
<td>Called by the agent to notify the reactor of new observations of external</td>
</tr>
<tr>
<td></td>
<td>timelines, with this information the state of dispatched or active goals is</td>
</tr>
<tr>
<td></td>
<td>updated. Calls notify of each available reactor for the observations</td>
</tr>
<tr>
<td></td>
<td>received to be handled by the new defined reactor.</td>
</tr>
<tr>
<td>bool doHasWork();</td>
<td>Called by the agent to know if there is pending work to perform in this</td>
</tr>
<tr>
<td></td>
<td>reactor. Calls hasWork method of each available reactor.</td>
</tr>
<tr>
<td>void step();</td>
<td>Called if doHasWork returns true. Handles the increment of the steps of</td>
</tr>
<tr>
<td></td>
<td>current deliberation and calls resume.</td>
</tr>
</tbody>
</table>

Table 8-3: Methods of Teleoreactor to be overloaded / defined when creating a new reactor

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handleInit();</td>
<td>This method is called after complete reactor initialization and can be used</td>
</tr>
<tr>
<td></td>
<td>for extra initialization before starting the reactor execution.</td>
</tr>
<tr>
<td>handleTickStart();</td>
<td>This method is usually used to post goals to other timelines.</td>
</tr>
<tr>
<td>notify(Observation);</td>
<td>This method is usually used to handle received observations.</td>
</tr>
<tr>
<td>boolean synchronize();</td>
<td>This method is usually used to post this reactors observations.</td>
</tr>
<tr>
<td>void handleRequest(goal_id);</td>
<td>This method is intended to process a received goal.</td>
</tr>
<tr>
<td>void handleRecall(goal_id);</td>
<td>This method is intended to process the cancellation or retrieval of a given</td>
</tr>
<tr>
<td>boolean hasWork();</td>
<td>This method is used to let the agent know if the reactor has work to do.</td>
</tr>
<tr>
<td>void resume();</td>
<td>This method performs the operations when hasWork returns true.</td>
</tr>
</tbody>
</table>

Each reactor is defined by a set of configurable parameters in a configuration file. An example generic file can be seen in Figure 8-3 and a real example file in Figure 8-4.

**XML CONFIGURATION OF A REACTOR**

```xml
<__ReactorClass__ name="__name__"
    lookahead="_#_"
    latency="_#_"
    <Internal name="__timeline1__"/>
    <Internal name="__timeline2__"/>
    <External name="__timeline3__"/>
    <External name="__timeline4__"/>
</__ReactorClass__>
```

Figure 8-3: XML generic configuration of a reactor

The parameters that need to be set are:

- __ReactorClass__ : The name of the class (a son class of Teleoreactor with all the needed methods) we want the new reactor to be.
- Name __name__ : The name the reactor will have. There can be different reactor made of the same class with different parameters, and the name is the parameter used to tell them apart.
Lookahead _#_: The reactor will only receive the goals with a start lower boundary between 
t0 (actual tick) + Lookahead. For reactive reactor, the lookahead should be 1, only accepting 
immediately goals.

Latency _#_: Time in ticks the reactor needs to deliberate. The goals won't start until t0 + 
lateness. This should be 0 for reactive reactor and the time to deliberate for deliberative 
reactor.

Internal Timeline name __timeline1__ or __timeline2__: Name for each of the internal 
timelines owned by this reactor.

External timeline name __timeline3__ or __timeline4__: Name for each of the external 
timelines subscribed to by this reactor.

More parameters can be added to configure the new reactor if needed.

---

**XML CONFIGURATION OF A REACTOR EXAMPLE**

```xml
<GciReactor name="GCI" latency="0" lookahead="1" log="1" tm="1" tmPeriod="4" level="E3"
timelinesFile="orbital_timeline_config.cfg">
  <External name="battery"/>
  <External name="roboticarm"/>
  <External name="mission"/>
  <External name="planner"/>
</GciReactor>

<DeliberativeReactor name="plannerReactor" latency="20" lookahead="200" log="1" addr="localhost"
port="4441">
  <Internal name="mission"/>
  <Internal name="planner"/>
  <External name="battery"/>
  <External name="roboticarm"/>
</DeliberativeReactor>
```

**Figure 8-4: XML example reactor configuration**

---

**8.1.2. GCI PARTICULARIZATION AND USE**

The Ground Control Interface Reactor is designed as a generic reactor. The particular information 
is loaded from the timeline configuration file. As explained in section 8.1.1.2 Define the timelines 
needed, this file defines all the timelines, its predicates and attributes, telecommands, telemetry 
codes, etc.

The path to this file is set in the reactor configuration, in the timelinesFile parameter. It is relative 
to the cfg path, the file should be inside the folder pointed by the environment variable 
$TREX_HOME/cfg/.

**8.1.2.1. AUTONOMY LEVEL TELECOMMANDS**

The autonomy level files, **TC_autonomy.dat**, are a special type of telecommand, used to change 
the autonomy level. This files will be processed in all the autonomy levels and have an immediate 
effect.

**8.1.2.2. E1: TELECOMMANDS**

The E1 telecommands are send via **TC_E1.dat** files, copied in the $TC_DIR folder. This files will be 
processed when the autonomy level is E1-E2-E3, and not in E4 level. This commands will be 
processed and executed immediately. The GCI uses all the information in the timeline_configuration_file to correctly constructs and send the desired goal. A generic TC_E1.dat and an example can be seen in the Table 8-4.
Table 8-4: TC E1 Example file

<table>
<thead>
<tr>
<th>GENERIC TC E1 FILE</th>
<th>EXAMPLE TC E1 FILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERGO <strong>TCcode</strong></td>
<td>ERGO CRAP</td>
</tr>
<tr>
<td><strong>param1</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>param2</strong></td>
<td>1.0 0.0 0.0</td>
</tr>
</tbody>
</table>

The configurable parameters of the telecommands are:

- __TCcode__: The code of the telecommand to execute. This code has to be defined in the timeline_configuration_file.
- __param1__, __param2__, etc: The parameters of the goal. This parameters have to be set in the order they were defined in the timeline_configuration_file. For composed parameters (like point3), the different values are separated by blank spaces. The parameters that have a default and a lastObs value set, can be left empty. It is important that there are no extra new line at the end, as it would be treated as a missing parameters.

8.1.2.3. E2: TIMETAG TELECOMMANDS

The E2 telecommands are sent via TC_E2.dat files, copied in the $TC_DIR folder. This files will be processed when the autonomy level is E2 and E3, and not in E1 or E4. This commands will be process in the time set in the file. The GCI uses all the information in the timeline_configuration_file to correctly constructs and send the desired goal. The format is the same of the E1 telecommands with the addition of the timetag. A generic TC_E2.dat and an example can be seen in the Table 8-5.

Table 8-5 TC E2 Example file

<table>
<thead>
<tr>
<th>GENERIC TC E2 FILE</th>
<th>EXAMPLE TC E2 FILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERGO <strong>timetag</strong></td>
<td>ERGO 2018.268.9.30.6</td>
</tr>
<tr>
<td><strong>TCcode</strong></td>
<td>CRAP</td>
</tr>
<tr>
<td><strong>param1</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>param2</strong></td>
<td>1.0 0.0 0.0</td>
</tr>
</tbody>
</table>

The configurable parameters of the telecommands are:

- __timetag__: The time the telecommand should be executed. The format of the timetag is YEAR.JULIAN DAY.HOUR.MINUTE.SECOND.
- __TCcode__: The code of the telecommand to execute. This code has to be defined in the timeline_configuration_file.
- __param1__, __param2__, etc: The parameters of the goal. This parameters have to be set in the order they were defined in the timeline_configuration_file. For composed parameters (like point3), the different values are separated by blank spaces. The parameters that have a default and a lastObs value set, can be left empty. It is important that there are no extra new line at the end, as it would be treated as a missing parameters.

8.1.2.4. E3: OBCP

OBCPs are a set of python scripts that can be executed and interact with ERGO. This scripts can get the status of the timelines and send goals. OBCP execution telecommands will only be processed on E3 autonomy level.

8.1.2.4.1. EXECUTING AN EXISTING OBCP

To execute an existing OBCP, a TC_E3_OBCP.dat file has to be copied into the $TC_DIR folder. This folder contains all the telecommands file that are to be processed when the autonomy level is the correct one. A generic example of this file can be seen in the Table 8-6. The number __#__ represent the code of the OBCP we want to execute. The available OBCP can be seen in the file code.mpy.h in the $ERGO_OBCP/ ergoOBCP/obcps/ folder.
### Table 8-6: Generic and example of a TC_E3_OBCP.dat file

<table>
<thead>
<tr>
<th>OBCP GENERIC TC FILE</th>
<th>OBCP EXAMPLE TC FILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERGO <strong>#</strong></td>
<td>ERGO 10</td>
</tr>
</tbody>
</table>

The OBCPs are scenario dependent, as they use the specific timelines names defined in the use case.

#### 8.1.2.4.2. CODING A NEW OBCP

To code a new OBCP, there are a set of interface functions to communicate the OBCPs and ERGO. This functions are shown in the Table 8-7.

#### Table 8-7: Available interface functions OBCP-ERGO

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool user.ERGOexecuteTC(string TC)</td>
<td>This method executes a TC. It uses the TC_code set in the timeline configuration file. The separation between attributes has to be &quot;;&quot;. It returns true.</td>
</tr>
<tr>
<td>String user.ERGOgetTimelineState(string timeline)</td>
<td>This methods is used to get the last observation received in a timeline. The name of the timeline is the one set in the timeline_configuration_file. It returns &quot;-1&quot; if the timeline was not found, &quot;default&quot; if no observation has been received, and the observation as a string otherwise.</td>
</tr>
<tr>
<td>Int user.ERGOisTimelineStopped(string timeline)</td>
<td>This method return whether the timeline is active or not. It used the information about the activePreds and inactivePreds in the timeline_configuration_file. It returns &quot;-1&quot; if the timeline name was not found, &quot;1&quot; if it is at an inactive status and &quot;0&quot; if it is at an active status.</td>
</tr>
<tr>
<td>bool user.ERGOIsAllStoppedFunction()</td>
<td>This method is used to know the general state of the whole system. It used the information of all the available timelines defined in the configuration files. It returns &quot;true&quot; if all the timelines are stopped and &quot;false&quot; otherwise.</td>
</tr>
</tbody>
</table>

All the OBCP have to be located in the $ERGO_OBCP/ ergoOBCP/obcps/ folder. An example of a simple OBCP can be seen in the Figure 8-5: OBCP Example file for the planetary case.

This OBCP consists of waiting until a valid observation of the robotic arm has been received, and then checking its value. If it is not at a home position, it sends it to home and waits for the operation to finish. It is important to note that every OBCP has an unique name, set at the end of the file: a = ObcpTest("OBCP_B07"). This name cannot be longer than 9 digits.
class ObcpTest(Obcp):
    
    def main(self):
        waitTime = 0.2

        # Wait for a valid observation.
        result = user.ERGOgetTimelineState("roboticarm")
        print("\033[1;36m[OBCP] Waiting for a valid observation of the robotic arm. \033[0m")
        #print("\033[1;36m[OBCP] result: ", result, " \033[0m")
        while result.find("default") != -1:
            #print("\033[1;36m[OBCP] result: ", result, " \033[0m")
            self.obcp_wait_interval_time(waitTime)
            result = user.ERGOgetTimelineState("roboticarm")
            print("\033[1;36m[OBCP] timeline obs received ", result, " \033[0m")

        # Check the status. If it is not at home, move to home.
        if result.find("\[X=-0.68,Y=-0.13,Z=0.80\]") != -1:
            print("\033[1;36m[OBCP] The robotic arm is at a home position. No need to move \033[0m")
        else:
            print("\033[1;36m[OBCP] The robotic arm is not at a home position. Moving to home \033[0m")
            
            #send to home
            user.ERGOexecuteTC("CRAM;;-0.68 -0.13 0.8")
            
            #wait for it to stop
            stopped = user.ERGOisTimelineStopped("roboticarm")
            print("\033[1;36m[OBCP] Waiting for the robotic arm timeline to go to idle \033[0m")
            while stopped != 1:
                self.obcp_wait_interval_time(waitTime)
                stopped = user.ERGOisTimelineStopped("roboticarm")
                print("\033[1;36m[OBCP] Robotic arm timeline is stopped \033[0m")
            
            print("\033[1;36m[OBCP] OBCP Finished \033[0m")

    a = ObcpTest("OBCP_B07")
    a.run()

Figure 8-5: OBCP Example file for the planetary case

Once the OBCP has been coded, it has to be added to the scripts that perform the translation to a lower level code. It is easily done by adding the name of the file to the coding.sh file existing in the same folder, in the obcpScripts vector.

Once it is added in the correct place, it will be translated and included in the code.mpy.h file so it can be called. This translation in performed every time the agent is build, there is no more intervention needed.

8.1.2.5. E4: High Level Goals

The E4 telecommands are sent via TC_E4.dat files, copied in the $TC_DIR folder. This files will be processed when the autonomy level is E4, and not in the rest of the autonomy levels. This commands will be processed immediatly, and the goals will be generated. The goal is sent in xml format and is parsed in the goal constructor. A generic TC_E4.dat and an example can be seen in the Table 8-8 and Table 8-9 respectively.
The configurable parameters of the telecommands are:

- **timelineName**: The name of the timeline we want to send the goal to.
- **predicateName**: The name of the predicate of the goal.
- **param1, param2**, etc.: The name of the parameter that want to be set in the goal.
- **type1, type2**, etc.: The name of the type of data used in the parameter. The available types and their attributes can be seen in the Table 8-1.
- **attType1, attType2**, etc.: the name of the attribute in the parameter.
- **value**: The value of the attribute in the parameter.

There are 4 parameters that all the goals need to have. These parameters are:

- **Start**: Time boundaries for the start time of the goal.
- **Duration**: Time boundaries for the duration time of the goal.
- **End**: Time boundaries for the end time of the goal.
- **Priority**: Priority of the goal.

<table>
<thead>
<tr>
<th>Table 8-8 Generic TC E4 file</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC E4 GENERIC FILE</td>
</tr>
<tr>
<td>&lt;?xml version=&quot;1.0&quot; encoding=&quot;utf-8&quot;?&gt;</td>
</tr>
<tr>
<td>&lt;Agent name=&quot;ERGO&quot;&gt;</td>
</tr>
<tr>
<td>&lt;Goal on=&quot;__timelineName__&quot; pred=&quot;__predicateName__&quot;&gt;</td>
</tr>
<tr>
<td>&lt;Variable name=&quot;__param1__&quot;&gt;&lt;<strong>type1</strong> <strong>attType1</strong>=&quot;<strong>value</strong>&quot; <strong>attType2</strong>=&quot;<strong>value</strong>&quot;/&gt;&lt;/Variable&gt;</td>
</tr>
<tr>
<td>&lt;Variable name=&quot;__param1__&quot;&gt;&lt;<strong>type1</strong> <strong>attType1</strong>=&quot;<strong>value</strong>&quot; <strong>attType2</strong>=&quot;<strong>value</strong>&quot; /&gt;&lt;/Variable&gt;</td>
</tr>
<tr>
<td>&lt;Variable name=&quot;start&quot;&gt;&lt;int min=&quot;__value__&quot; max=&quot;__value__&quot; /&gt;&lt;/Variable&gt;</td>
</tr>
<tr>
<td>&lt;Variable name=&quot;duration&quot;&gt;&lt;int min=&quot;__value__&quot; max=&quot;__value__&quot; /&gt;&lt;/Variable&gt;</td>
</tr>
<tr>
<td>&lt;Variable name=&quot;end&quot;&gt;&lt;int min=&quot;__value__&quot; max=&quot;__value__&quot; /&gt;&lt;/Variable&gt;</td>
</tr>
<tr>
<td>&lt;Variable name=&quot;priority&quot;&gt;&lt;int min=&quot;__value__&quot; max=&quot;__value__&quot; /&gt;&lt;/Variable&gt;</td>
</tr>
<tr>
<td>&lt;/Goal&gt;</td>
</tr>
<tr>
<td>&lt;/Agent&gt;</td>
</tr>
</tbody>
</table>

Table 8-9 Example of a TC E4 file

<table>
<thead>
<tr>
<th>Table 8-9 Example of a TC E4 file</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC E4 EXAMPLE FILE</td>
</tr>
<tr>
<td>&lt;?xml version=&quot;1.0&quot; encoding=&quot;utf-8&quot;?&gt;</td>
</tr>
<tr>
<td>&lt;Agent name=&quot;ERGO&quot;&gt;</td>
</tr>
<tr>
<td>&lt;Goal on=&quot;mission&quot; pred=&quot;traversing&quot;&gt;</td>
</tr>
<tr>
<td>&lt;Variable name=&quot;topose&quot;&gt;&lt;point3 X=&quot;8&quot; Y=&quot;-1&quot; Z=&quot;1&quot;/&gt;&lt;/Variable&gt;</td>
</tr>
<tr>
<td>&lt;Variable name=&quot;start&quot;&gt;&lt;int min=&quot;1&quot; max=&quot;50&quot; /&gt;&lt;/Variable&gt;</td>
</tr>
<tr>
<td>&lt;Variable name=&quot;duration&quot;&gt;&lt;int min=&quot;10&quot; max=&quot;100&quot; /&gt;&lt;/Variable&gt;</td>
</tr>
<tr>
<td>&lt;Variable name=&quot;end&quot;&gt;&lt;int min=&quot;11&quot; max=&quot;150&quot; /&gt;&lt;/Variable&gt;</td>
</tr>
<tr>
<td>&lt;Variable name=&quot;priority&quot;&gt;&lt;int min=&quot;10&quot; max=&quot;10&quot; /&gt;&lt;/Variable&gt;</td>
</tr>
<tr>
<td>&lt;/Goal&gt;</td>
</tr>
<tr>
<td>&lt;/Agent&gt;</td>
</tr>
</tbody>
</table>
8.1.3. CMD REACTORS PARTICULARIZATION

CMDReactors are a special type of reactive reactor intended to interface with the FL of the system. These reactors are in charge of handling the communication of goals and observations between the controller and the functional layer.

CMDReactors have been designed to be a generic reactor which can be adapted to work with different types of timelines and datatypes and to be connected with any FL platform. All CMDReactors are instantiated using the same ERGOPlugin (shared library) and are configured with the information needed (timelines and type of timelines) during initialization from the configuration file (examples are provided in sections A.2 and B.2).

It is important to understand how this reactor handles the timelines associated to it. Note that this type of reactor will only handle internal timelines are separated in two different types (see representation and differences in Figure 8-6):

- **Action timelines**: These timelines are the ones that can receive goals from the controller and dispatch them to the FL. They will update the information of the action generating observations with the information provided from the FL.
- **State timelines**: These timelines only represent the status of a component of the FL. This means that this type of timeline is not able to receive nor handle goals, just process information from FL and post it to the controller as observations.

![Figure 8-6 CMDReactor Timeline types](image)

These (CMDreactors) do not accept being subscribed to external timelines as it is expected that any information needed is received from the FL, not from any other reactor (through the agent controller).

CMDreactors cannot post goals in any other reactor, as reactive executive reactors just process goals downwards from the controller to the FL and observations upwards from FL to the controller. This type of reactor executes goals as they arrive if valid, but does not store goals nor accumulate them.

Goals and observations only flow in the directions represented in Figure 8-6.
8.2. GODA COMPONENT

The Goal Orientated Data Analysis (GODA) Component is responsible for processing data from the perception system and generating new candidate goals as input to re-planning activities. This provides a necessary mechanism to generate system goals autonomously, supporting the high level ERGO objective of highly autonomous systems development. These goals could vary from flagging particular data as pertinent, to directing attention of higher resolution imagers to capture serendipitous science, or perhaps triggering re-planning in order to acquire images from a better position. The key intuition is that intelligent analysis of the environment the system finds itself in may lead to new goals and actions to be taken, which require immediate action and therefore do not allow ground-in-the-loop operations.

Figure 8-7: GODA design overview

The figure above shows there are seven main elements to the GODA system. Of these, the Saliency map, the Classifier and the Goal Generator components form the core. The Saliency Map component is designed to segment the image into regions of interest, which the Classifier then classify as known labels. The Goal Generator finally maps from these detections to specific goals (for the planner) with attached metrics to evaluate. The API forms the interface between GODA and the rest of the system and also manages the operation of the core components of GODA.

The Mappings, Models and Configurations files provide the adaptability for the system to different use cases. The models contain the learnt parameters for machine-learning based vision systems implemented in the Classifier. Training these is a computationally expensive offline process, but allows for the on-board detector to be higher performance and easily adapted to different environments. Similarly, the mapping files manage the mapping between detections, goals and metrics and so allows for re-configuration of the system to suit different objectives. The configurations files are general configuration parameters used by the GODA API to configure each of the components. They are used to abstract the implementation of the functions from any hard coded configuration values and allow for easy parametrisation if needed. Next sections describe in more detail how to install, configure and run the GODA component.
8.2.1. GODA TAILORING

8.2.1.1. DIRECTORY STRUCTURE
Once you unpack the GODA distribution code you should have the following directories:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERG_GODA</td>
<td>Main directory where GODA is unpacked</td>
</tr>
<tr>
<td>ERGO_GODA/bin</td>
<td>Build directory for the example executable and unit tests executables</td>
</tr>
<tr>
<td>ERGO_GODA/lib</td>
<td>Directory for all the compiled libraries</td>
</tr>
<tr>
<td>ERGO_GODA/models</td>
<td>SCISYS Machine Learning models</td>
</tr>
<tr>
<td>ERGO_GODA/obj</td>
<td>Object files for compilation</td>
</tr>
<tr>
<td>ERGO_GODA/src</td>
<td>Main directory with the GODA source code</td>
</tr>
<tr>
<td>ERGO_GODA/test</td>
<td>Unit tests for the GODA library</td>
</tr>
<tr>
<td>ERGO_GODA/usage_example</td>
<td>Example on how to run GODA</td>
</tr>
</tbody>
</table>

8.2.1.2. SYSTEM REQUIREMENTS AND INSTALLATION
To run the GODA component as tested, the following requirements must be met:

- Ubuntu 16.04 64bit
- GCC 5.4.0-6 ~ 20160609
- OpenCV 3.1.0 compiled for Ubuntu 16.04 x86_64
- libssl-dev package installed
- TinyXML-2 Library
- CppUnit Testing Library

The following environment variables must be set:

1. export ERGO_GODA_HOME=/directory/path/to/ERGO_GODA
2. export LD_LIBRARY_PATH=$ERGO_GODA_HOME/lib:$ERGO_GODA_HOME/lib/scisys-ml-models

Where "/directory/path/to/ERGO_GODA” is the main directory where the unpacked GODA distribution code is.

To compile the GUI version of the GODA library:

1. cd $ERGO_GODA_HOME
2. make clean GUI

Or to compile the headless version of the GODA library to be used in a system then:

1. cd $ERGO_GODA_HOME
2. make clean all
To run the unit tests and make sure that GODA has compiled correctly and the requirements are met:

1. cd $ERGO_GODA_HOME/bin
2. ./test_{test_name}
8.3. MISSION PLANNER

8.3.1. REACTOR

The Mission Planner Reactor is responsible for initialising and calling the Stellar planner, parsing TrexToken goals and observations and PDDL to and from intermediate data structures, and dispatching goals and checking the current state against expected states. It also handles situations where re-planning is required, such as unexpected situations and the receipt of new goals.

![Mission Planner Reactor](image)

**Figure 8-8: Mission Planner components**

The above figure shows the main components of the Mission Planner Reactor:

- **Reactor**: This component is the main interface called by the deliberative reactor. It initialises and calls the other components listed below using PDDL domain and problem files defined in its configuration file, and provides the logic for triggering planning and re-planning.
- **Observer**: Checks incoming observations against the states expected at the current transitions in the plan. Triggers re-planning if the plan is no longer valid.
- **Dispatcher**: Provides the current output goals of the plan, given the current execution time.
- **Wrapper**: Provides an interface between the reactor and planner components, along with performing translations between TrexToken and PDDL data structures.

8.3.1.1. DIRECTORY STRUCTURE

The mission planner reactor has the following directories:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deliberativefunc</td>
<td>Main directory which contains the mission planner reactor</td>
</tr>
<tr>
<td>deliberativefunc/common</td>
<td>Directory containing files used in most classes in the reactor</td>
</tr>
<tr>
<td>deliberativefunc/reactor</td>
<td>Directory for the reactor class</td>
</tr>
<tr>
<td>deliberativefunc/wrapper</td>
<td>Directory for the planner wrapper class and functions for initialising and calling the planner</td>
</tr>
<tr>
<td>deliberativefunc/translator</td>
<td>Directory contains the classes that deal with parsing PDDL files and translating to/from PDDL and TrexTokens</td>
</tr>
</tbody>
</table>
Directory | Description
---|---
deliberativefunc/domains | Directory with PDDL domain and Problem files for defining the scenarios
deliberativefunc/executive | Contains the classes responsible for checking the current system state with the expected state, and for correctly selecting transitions for dispatch
deliberativefunc/tests | Directory with the unitary tests for checking reactor functionality is correct

8.3.1.2. SYSTEM REQUIREMENTS AND INSTALLATION

To run the Mission Planner component needs the same requirements as those defined for the agent, in order to run. In addition, it requires the pthread library and the C++11 standard.

To compile the Mission Planner Reactor and Stellar within ERGO:
- Run 'source build_make' in the Agent/agent directory of ERGO

8.3.2. STELLAR

The Mission Planner (Stellar) Component is responsible for generating a plan of actions. Stellar receives the information divided into two files: the problem file that describes the state of the world and the goals to be achieved; and the domain file that describes the actions that can be performed in the environment by the rover or the orbital. This input information is simplified and translated into SAS+, a much more compact language based on numerical variables. This representation is used by the search algorithm and the different heuristics to generate a solution plan.

The figure above shows the two main modules of Stellar Search component. The PDDL2SAS is the translator system, which translates PDDL input (Domain and Problem) into SAS+ language. The search component is the main component of the Stellar Search. This component is divided into six subcomponents: State Management, Action Application Management, STN Solver, Search and Heuristic. This module runs a heuristic search algorithm using the delete relaxation heuristic (FF heuristic) to guide the search and Simple Temporal Network (STN) to check that the temporal constraints are fulfilled by the plan generated during search. This module is described in detail in the Final Design Report. The Domain and Problem are PDDL files and the output file is SAS+ representation of the combination of both PDDL files. The output is a plan of actions.

Figure 8-9: Stellar Design Overview
8.3.2.1. DIRECTORY STRUCTURE

The mission planner stellar have the following directories:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stellarsearch</td>
<td>Main directory where stellarsearch is unpacked</td>
</tr>
<tr>
<td>stellarsearch/pddl2sas</td>
<td>Main directory for the translator from PDDL to SAS+</td>
</tr>
<tr>
<td>stellarsearch/search</td>
<td>Main directory for the search</td>
</tr>
<tr>
<td>stellarsearch/external_functions</td>
<td>Main directory with the external functions libraries (.so files)</td>
</tr>
<tr>
<td>stellarsearch/search/search_engines</td>
<td>Directory with the search engines source code</td>
</tr>
<tr>
<td>stellarsearch/search/plugins</td>
<td>Directory with the plugins system source code</td>
</tr>
<tr>
<td>stellarsearch/search/includes</td>
<td>Directory with the header files included when the planner is compiled as a library.</td>
</tr>
<tr>
<td>stellarsearch/search/heuristic</td>
<td>Directory with the different implementations of the heuristic functions</td>
</tr>
<tr>
<td>stellarsearch/tests</td>
<td>Directory with the different test in PDDL</td>
</tr>
</tbody>
</table>

There are some more directories inside search folder in which there are stored the different implementation of open and close list, options handler, planning tasks and utils.

8.3.2.2. SYSTEM REQUIREMENTS AND INSTALLATION

To run the stellar planner component as tested in standalone mode, the following requirements must be met:

- Ubuntu 16.04 64bit
- GCC 5.4.0-6 ~ 20160609
- Linux libraries to compile stellar
  - g++-multilib
  - make
  - python

To compile Stellar within ERGO:
- Run ‘source build_make’ in the Agent/agent directory of ERGO

To compile the standalone version of Stellar:
- cd stellarsearch
- make

To run stellar search and use any of the tests:
- cd stellarsearch
9. FUNCTIONAL LAYER DEVELOPMENT

9.1. ROVER GUIDANCE

The structure and functionality of Rover Guidance library (RG) and its interfaces are presented in Final Design Report [AD.12], sections 5.2.2-5.2.4. This highlights the library interface functions, the input and output commands and datasets required by the RG as well as interpretation of module states. Those sections summarise also RG functions sequencing, required call frequencies and internal interfaces for multi-threaded implementation.

The public RG interfaces are self-contained within the RG component named RoverGuidance (that can be found in RG/SourceCode/GncAlgorithms/RoverGuidance). The catalog structure of RoverGuidance is presented in Figure 9-1. For external user (using RG in 'blackbox' approach) RoverGuidance/PublicFunctions/RoverGuidance_PublicFunctions.h header is providing full interface calls within RG. The rest of important definitions (data structures, commands and states) can be found in RoverGuidance/ConstantDefs and RoverGuidance/DataStructDefs.

![Figure 9-1: RG directory structure](image)

An example of how to call the interface functions is provided as a test harness named InterfaceSkeletonTest which can be found in RG/SourceCode/TestHarnesses. To run the provided test harness, first the environment and folder structure must be prepared and the application must be built: this is done by the provided script named BuildTargetErgoRg (requires CMake 2.8+). After the build process is finished, test harness can be run using RunInterfaceSkeletonTest. Both scripts provide step by step instructions of building and running the application using RG.

During runtime, RG outputs several data files, as well as a log output to screen, which can be used to monitor the status and progress during a run. An example of such log can be seen in Table 9-1 below.
Table 9-1: Example of RG log output

<table>
<thead>
<tr>
<th>Log Output RG (Example)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[RGI] NavModeSelection_init(): Initialising NavModeSelection</td>
</tr>
<tr>
<td>[RGI] PlanML_init(): Initialising Plan ML</td>
</tr>
<tr>
<td>[RGI] PlanML_loadOrbitalNavMap(): Loading Orbital NAVMap...</td>
</tr>
<tr>
<td>[RGI] loadParamDouble(): NavMap:cellSize_m not in .ini file.</td>
</tr>
<tr>
<td>[RGI] IO_NavImg_loadfileSet(): Loaded Image:</td>
</tr>
<tr>
<td>[RGI] PlanML_loadOrbitalDem(): Loading Orbital DEM...</td>
</tr>
<tr>
<td>[RGI] ResourcesEstimation_init(): Initialising Resources Estimation</td>
</tr>
<tr>
<td>[RGI] NavL2_init(): Initialising Nav L2</td>
</tr>
<tr>
<td>[RGI] HazardPrevention_init(): Initialising HP (Hazard Prevention)</td>
</tr>
<tr>
<td>[RGI] PlanML_do(): ******************** Entering Plan ML session 0 ****************</td>
</tr>
<tr>
<td>[RGI] PlanML_do(): Archive Index 0</td>
</tr>
<tr>
<td>Exiting Rover Guidance Interface</td>
</tr>
<tr>
<td>[RGI] NavL2_exit(): Exiting NavL2</td>
</tr>
<tr>
<td>[RGI] CutLocalDem_init(): Terrain slice is of size 601</td>
</tr>
<tr>
<td>[RGI] CutLocalDem_cut(): Archive Index 0</td>
</tr>
</tbody>
</table>

9.1.1. REAL-TIME VISUALISATION TOOL

The tool called `displayRealTimeNavAndPathData` can be used to visualise latest data processed by RG. It loads archived data form `OutputData` folder so too work properly it requires that the data archiving is enabled. The application can be used during runtime (will refresh with latest data available) or post-run (will display only the latest available states).

This standalone application can display:

- latest generated orbital NavMap with planned long term path & planned drive path;
- latest generated local NavMap with planned drive path (if L2 is in use);
- the application refreshes the figures, if during a test run, new data is written to RG’s `OutputData` folder.

**Prerequisite and installation:**

Version delivered is compatible with 64bit Linux systems only (tested in CentOS 6.9). It requires MATLAB runtime environment (2013b) to be installed, which is free to use and can be downloaded from official MATLAB webpage:

- [https://uk.mathworks.com/products/compiler/matlab-runtime.html](https://uk.mathworks.com/products/compiler/matlab-runtime.html)

After installation on the target computer, the `LD_LIBRARY_PATH` and `XAPPLRESDIR` environmental variables should be configured, according to the readme file provided by the application.

**Usage:**

To start the application call with the following arguments (the application can be anywhere on the drive):

```
./run_displayRealTimeNavAndPathData.sh $LD_LIBRARY_PATH Linux NavWithPath 0 1 path /yourFolder/TestName/OutputData
```

```
./run_displayRealTimeNavAndPathData.sh $LD_LIBRARY_PATH Linux NavOnly 0 1 path /yourFolder/TestName/OutputData
```

If data is not available, application displays empty windows. The following figure presents examples of windows to be expected while data is available:
Figure 9-2 Example representing the output of real time visualisation tool

The Table 9-2 represents color-coding for specific data products:

<table>
<thead>
<tr>
<th>Visualisation tool</th>
<th>Colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local path:</td>
<td>magenta</td>
</tr>
<tr>
<td>Global path:</td>
<td>black</td>
</tr>
<tr>
<td>Traversable terrain on Global map:</td>
<td>blue, green, orange, red (L1-L4 respectively)</td>
</tr>
<tr>
<td>Unknown terrain</td>
<td>dark blue</td>
</tr>
<tr>
<td>Forbidden terrain on both Local/Global map:</td>
<td>white</td>
</tr>
<tr>
<td>High cost terrain both Local/Global maps:</td>
<td>red-brown</td>
</tr>
<tr>
<td>Point turns:</td>
<td>magenta cross in a circle</td>
</tr>
</tbody>
</table>

All distances/coordinates units are given in meters [m].

9.2. ROBOTIC ARM TAILORING

CollisionChecker: The Robotic Arm library uses the URDF model of the arm in order to get the arm morphology to check collisions and compute the kinematics and to get the joint limits, so a precise URDF model has to be provided.

Some examples of models are given in robot_description. In CollisionChecker module, in the configuration files, the path to the URDF model of the robot has to be provided, in case there are pairs of links that doesn’t need collision checking, because they are adjacent or they never collide, a list can be provided to improve performance and avoid false positives. A “black_list” of critical robot configurations that should be avoided can also be provided, and in case the URDF defines not just a robotic arm but a bigger robot with an arm, the root_link of the arm and its end_effector must be provided to avoid taking into account the extra joints of the robot in the planning and to add them to the environment an avoid colliding the arm with the robot.
**Driver:** For each specific hardware, there has to be a driver that executes the desired functionality reading information and commanding movements. There is an abstract interface to use with the rest of the code in the Driver module. There are some functions that are mandatory for the basic functionality of the library, and some of them are optional.

A driver class has to be developed that inherits from Driver and implements the functions with "= 0" in the definition. The optional functions have a default implementation that just informs the user that are not implemented. Note that rather move() or executeTrajectory() have to be implemented, if you just implement move(), executeTrajectory() will execute that function every time step, and in case the needed implementation is more complex than that, a custom executeTrajectory() function can be implemented.

**Camera:** This module is just in case there is a camera on the tip of the arm and a marker detection has to be done. If the camera is compatible with OpenCV VideoCapture (driver v4l2), or it is an IDS Ueye camera, there is an initialization and capture method implemented, otherwise it has to be provided.

The markers to be detected are defined in data/markers. There are a default dictionary, but other one can be generated and substitute the current one. The files in markers and boards folders are just for printing the desired markers to use them, and the files in targets are lists of the markers included in a certain object to detect and the position of the corners of each marker with respect to the reference frame of the object. The module will provide the pose of the reference frame of the object in the camera frame.

**Manager:** A manager class can be developed for a certain use case encapsulating the interactions between the modules of the library in simpler commands.

In case the camera module is used, there must be a hand-eye calibration that stores the pose of the camera frame with respect to the end effector of the arm to be able to use the camera module information.

### 9.3. ADDING ADDITIONAL LIBRARIES

To make a new specific component for the functional layer, it should be implemented as a class providing the desired functionality in public methods and compiles as a shared library, that then would be linked by the executable (x86_functional) that uses rest of functional layer components (as libraries) or by the manager of a functional layer component that encapsulates the functionality for a certain use case.
10. AGENT & FUNCTIONAL LAYER INTEGRATION

To test the adaptability of the system there are two scenarios proposed to be instantiated: orbital and planetary scenario.

Before and in depth description of both implementation some general remarks are given about naming convention followed to integrate TREX Agent within the Agent TASTE function. These conventions are not compulsory but facilitates the integration process as it is demonstrated in the following section.

10.1. AGENT CONTROLLER INTEGRATION - GENERAL REMARKS

To make a smoother and more user-friendly integration between the TREX components and the TASTE function a set of naming conventions have been followed.

10.1.1. INTERFACES

Every TASTE function have a set of RI and PI that define the behavior of it and the connections with the rest of TASTE functions.

At this point and taking into account what is shown in Figure 10-1, Agent TASTE function have interfaces with two different purposes:

- **Behavior of Agent TASTE Function:**
  - DO_STEP (RI): This is a cyclic RI that ensures that the agent is called at the desired rate (remember that each Tick corresponds to any number of seconds). Inside this RI agent doNext() method is called to perform another cycle of the Agent.

- **Connection with other TASTE functions:** The interface between the Agent and the FL is made through the CMDReactors. These reactors have different interfaces to send and receive information from and to FL for each timeline; when integrated with TASTE this methods for each timeline are linked to TASTE interfaces; one for sending the goals and one for receiving state updates:
  - TIMELINENAME_STATUS (PI): Interface to receive information from FL. Note that the name of the PI follows the name convention shown: timeline name in uppercases, underscore and "request" in uppercases. These interfaces are used by both timeline types. It is recommendable to have a DO_STEP on each TASTE function to update the state using this TIMELINE_STATUS periodically.
  - TIMELINENAME_REQUEST (RI): Interface to send information to the FL. The name of the RI is equivalent to the PI name but finished "status" in uppercases. These interfaces (RI) are only for the AT.

![Figure 10-1 Example of RI and PI between TASTE and TREX components](image)
10.1.2. DATA TYPES

To be able to connect with the FL the data have to be translated into ASN1 types that flow through the TASTE interfaces. This data types follow the naming convention and parameters of TREX timelines. As an example a timeline configured with the following XML will be coded into an ASN1 type through the following:

- ASN1 data structure name comes from the timeline (first letter upper case) name plus status: Timelinenamestatus.
- ASN1 predicate is an enumeration that contains all the predicates (named as timelinename-predicate) plus a cancel predicate. This enumeration is named with the timeline name (first letter upper case) followed by “Pred”.
- ASN1 parameters are the union of all parameters of each predicate without repetition. They are named the same way as the parameters in the timeline configuration file.
- Parameters data type are TASTE types similar to the ones used in TREX (domains).

Figure 10-3: and Figure 10-4: serve as a concrete example with a timelines from the ERGO Orbital case. The relations previously described are applied to generate Figure 10-4: based on Figure 10-3: information.

Note that both RI and PI for each timeline use the same data type.
XML configuration

```xml
<Timeline name="roboticarm" tmname="TPRA" rname="RPRA">
  <inactivePred name="idleat" tcname="0">
    <parameter name="atposition" type="point3" default="lastObs"/>
  </inactivePred>
  <inactivePred name="picked" tcname="0">
    <parameter name="atposition" type="point3" default="lastObs"/>
  </inactivePred>
  <activePred name="moving" tcname="CRAM">
    <parameter name="fromposition" type="point3" default="lastObs"/>
    <parameter name="toposition" type="point3" default="needed"/>
  </activePred>
  <activePred name="picking" tcname="CRAP">
    <parameter name="sampleid" type="string" default="needed"/>
    <parameter name="fromposition" type="point3" default="lastObs"/>
    <parameter name="toposition" type="point3" default="needed"/>
  </activePred>
  <activePred name="dropping" tcname="CRAD">
    <parameter name="sampleid" type="string" default="needed"/>
    <parameter name="fromposition" type="point3" default="lastObs"/>
    <parameter name="toposition" type="point3" default="needed"/>
  </activePred>
  <errorPred name="fault" tcname="0"/>
</Timeline>
```

**Figure 10-3**: XML configuration of a timeline

ASN data

```plaintext
RoboticarmPred ::= ENUMERATED { roboticarm-idleat, roboticarm-picked, roboticarm-moving, roboticarm-picking, roboticarm-dropping, roboticarm-fault, roboticarm-cancel }

```

**Figure 10-4**: ANS1 data structures and enums based on a timeline

### 10.1.3. AUTOMATIC CODE GENERATION

Following such a structured and common naming convention for all the interfaces with each timeline results in a repetitive code with the same structure just to translate between Variable vectors to ASN1 attributes and predicates to ASN1 enumerations.

The essence of this code can be abstracted to be particularized for each case automatically. A python script tool has been develop to, parsing the configuration file, generate the required code to interface the TREX Agent Controller with the TASTE function and with the rest of the FL TASTE functions in consequence.

This tool has as an input the configuration file to parse (note that it has to reference the timeline configuration file somewhere on it) and the path where the output files will be located (usually the Agent Taste function src folder).
When execute it generates the following code files:

- **reactor_callbacks.h**: containing all the callback for CMDReactor.
- **agent_taste_adaptation.h**: The initialization of the agent and all the CMDReactors callbacks along with all the RI with the translation from ASN1 to Variable vector types and linked to each timeline.
- **asn_c_translation.h**: Coding the translation between predicates (Symbols) and ASN1 enum values.
- **Trex_to_taste.asn**: ASN1 data structures and enumerations do code all the information from the timelines. This is generated in the folder preceding the one given as input.
11. APPLICATION TAILORING

11.1. AGENT CONFIGURATION

11.1.1. AGENT MAIN CONTROLLER & REACTORS CONFIGURATION

An xml configuration’s file is used at start up by the agent to identify the reactors that are part of the system, and the interaction among them. Although this operation is the last step, we have decided to explain it in first place, because it helps the user to understand the architecture. The following figure shows an example of a configuration’s file for the agent:

```xml
<?xml version="1.0" encoding="utf-8"?>
<Agent name="ERGO" finalTick="10000">
  <!-- Normal reactors load their parameters via the TeleoReactor( xml_arg arg ) constructor. The following parameters are loaded by this constructor:
  - name
  - latency
  - lookahead
  - log
  - config (optional)
  and then for each timeline we check if its external or internal..
  External timelines have name, goals = 0 means no goals are posted in this timeline by this reactor.
  Internal timelines have name only

  Plugins CMDReactor and CMDReactor are concrete reactor types which also need predicates to be defined in configfile. -->
  <Plugin name="ERGOgcireactor"/>
  <Plugin name="ERGOdeliberative.reactor"/>
  <Plugin name="ERGOCMDReactor"/>
  <Plugin name="ERGOgodareactor"/>
  <Plugin name="ERGOvitre"/>

  <GciReactor name="GCI"
    latency="0"
    lookahead="1"
    log="1"
    tm="T"
    tmPeriod="4"
    level="E4"
    timelinesFile="planetary_timeline_config.cfg">
    <External name="camera"/>
    <External name="scientific.camera"/>
    <External name="camera"/>
    <External name="battery"/>
    <External name="antenna"/>
    <External name="guidance.cmd"/>
    <External name="guidance.plan"/>
    <External name="rover.position"/>
    <External name="robotic.arm"/>
    <External name="log"/>
    <External name="planner"/>
    <External name="mission"/>
    <External name="gods"/>
  </GciReactor>

  <GodaReactor name="GodaReactor"
    latency="0"
    lookahead="100"
    log="1"
    level="E4"
    camScanFreq="0.20"
    defaultNewGoalPriority="5"
```
<debugTraces="1"/>

<Internal name="gods" />
<External name="mission" />
<External name="roverposition" goals="0" />
<External name="scientificcamera" />
</GodReactor>

<DeliberativeReactor name="MissionPlannerReactor"
  latency="20"
  lookahead="200"
  log="1">
  <Internal name="planner" />
  <Internal name="mission" />
  <External name="roverposition" />
  <External name="roboticarm" />
  <External name="camera" />
  <External name="antenna" />
</DeliberativeReactor>

<CMDReactor name="CameraReactor"
  latency="0"
  lookahead="1"
  timeout="-1"
  timelineConfig="planetary_timeline_config.cfg">
  <Internal name="camera" />
  <Internal name="scientificcamera" />
</CMDReactor>

<CMDReactor name="BatteryReactor"
  latency="0"
  lookahead="1"
  timeout="-1"
  timelineConfig="planetary_timeline_config.cfg">
  <Internal name="battery" goals="0" />
</CMDReactor>

<CMDReactor name="AntennaReactor"
  latency="0"
  lookahead="1"
  timeout="-1"
  timelineConfig="planetary_timeline_config.cfg">
  <Internal name="antenna" />
</CMDReactor>

<CMDReactor name="GuidanceReactor"
  latency="0"
  lookahead="1"
  timeout="-1"
  timelineConfig="planetary_timeline_config.cfg">
  <Internal name="guidancecmd" />
  <Internal name="guidanceplan" goals="0" />
  <Internal name="roverposition" goals="0" />
  <Internal name="log" />
</CMDReactor>

<CMDReactor name="RampReactor"
  latency="0"
  lookahead="1"
  timeout="-1"
  timelineConfig="planetary_timeline_config.cfg">
  <Internal name="roboticarm" />
</CMDReactor>

<VitreReactor name="Vitre"
  latency="0"
  lookahead="0"
  log="0"
  port="31415"
  datadir="data" >
  <External name="camera" goals="0" />
  <External name="scientificcamera" goals="0" />
  <External name="gods" goals="0" />
  <External name="battery" goals="0" />
</VitreReactor>
Figure 11-1: Agent’s Configuration File

In Figure 11-1 we can see the layout of a configuration’s file. The first part (indicated in green) describes the agent’s name and additional parameters for the agent, like the time that the agent will run (finalTick). Plugins are dynamic libraries that will be loaded at start-up and each of them contains code for a specific reactor. Note that the granularity of a tick is an adjustable system parameter (for instance, it can be 0.1 seconds or 1 second) that will be defined during the detailed design phase.

Each of the lines in red describe a different reactor. In this configuration, we have a GCI, a planner reactor, a command dispatcher and a viewer (switchView). The “Viewer” reactor is completely passive; it only has external timelines that it is subscribed to, in order to show its evolution over time. The "goals=0" indicates that this external timeline will not be used to post goals.

Each reactor has its own latency and lookahead (i.e. deliberation horizon) that are parameters set in the configuration’s file.

In addition, each reactor has a set of external timelines and internal timelines. Internal timelines are those whose value is directly set by the reactor, meanwhile external timelines are those that are owned by other reactors. To set the value of an external timeline, we need to post a goal to the reactor that handles that timelines, using a specific function.

As we can see in this configuration, reactors conform a hierarchy. Reactors at a higher level receive high-level goals, and they decompose these goals into sub-goals that are posted to lower-level reactors. At the top of the hierarchy we find the Ground Control Interface. For instance, when the Ground Control Interface receives a telecommand in E4, (for instance “perform experiment A at position X, Y”) it posts this goal to the planner reactor, posting that goal to its mission timeline, which generates a plan. This plan includes lower level goals for the external timelines that are handled by the CmdDispatcher reactor, namely RobotBase (to go to the desired position) camera (to take an image), communication (to send the data to ground at a given time) etc....

The final configuration of ERGO will include additional reactors (the Robotic arm mission planner, Guidance, as well as GODA) but this example is provided for the user to understand the system’s architecture.

11.1.2. GODA CONFIGURATION

There are two main files that the GODA can be configured with. The general configuration file that contains parameters for the GODA API to configure each of the components and the mapping file that configures the mapping between detections, goals and metrics.

11.1.2.1. GENERAL CONFIGURATION

The general configuration file is passed to the GODA API to configure the rest of the GODA components. This file is an xml file with one main element called <GodaConfiguration>. Inside this element there are three other core xml elements - <GoalGeneratorConfiguration>, <ClassifierConfiguration> and <SaliencyConfiguration> - that are described in Table 11-2, Table 11-3 and Table 11-4.

An example of such a configuration file can be found in table Table 11-1 below.
Table 11-1: Example of the GODA general configuration file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<GodaConfiguration>
  <GoalGeneratorConfiguration>
    <EnabledGoals>7</EnabledGoals>
    <MappingFile>${ERGO_GODA_HOME}/bin/exampleGoalMapping.csv</MappingFile>
  </GoalGeneratorConfiguration>
  <ClassifierConfiguration>
    <Model>
      <Name>Rock Model</Name>
      <ModelPath>${ERGO_GODA_HOME}/models/scisys-ml-rock_model-fold_0.modelx</ModelPath>
      <Classes>
        <Class Label="rock" Enabled="1" />
      </Classes>
    </Model>
    <Model>
      <Name>Dunes Model</Name>
      <ModelPath>${ERGO_GODA_HOME}/models/scisys-ml-dunes_model-fold_0.modelx</ModelPath>
      <Classes>
        <Class Label="dunes" Enabled="1" />
      </Classes>
    </Model>
    <Model>
      <Name>Rocks Model</Name>
      <ModelPath>${ERGO_GODA_HOME}/models/scisys-ml-rocks_model-fold_0.modelx</ModelPath>
      <Classes>
        <Class Label="rocks" Enabled="1" />
      </Classes>
    </Model>
    <Model>
      <Name>Shadow Model</Name>
      <ModelPath>${ERGO_GODA_HOME}/models/scisys-ml-shadow_model-fold_0.modelx</ModelPath>
      <Classes>
        <Class Label="shadow" Enabled="1" />
      </Classes>
    </Model>
    <Model>
      <Name>Sky Model</Name>
      <ModelPath>${ERGO_GODA_HOME}/models/scisys-ml-sky_model-fold_0.modelx</ModelPath>
      <Classes>
        <Class Label="sky" Enabled="1" />
      </Classes>
    </Model>
  </ClassifierConfiguration>
  <SaliencyConfiguration>
    <ColorChannels>RGB</ColorChannels>
    <ResizeToInput>1</ResizeToInput>
    <SubtractMean>1</SubtractMean>
    <GaussianBlurSigma>0.045</GaussianBlurSigma>
    <MapWidth>64</MapWidth>
    <SegmentationThresholdMin>0.3</SegmentationThresholdMin>
    <SegmentationThresholdMax>1.0</SegmentationThresholdMax>
    <CropXMin>0</CropXMin>
    <CropWidth>0</CropWidth>
    <CropYMin>0</CropYMin>
    <CropHeight>0</CropHeight>
  </SaliencyConfiguration>
  <TestDataConfiguration>
    <Enabled>0</Enabled>
    <Result>
      <ClassLabel>rocks</ClassLabel>
      <BoundingBox X="80" Y="100" Width="250" Height="75" />
      <Confidence>0.8</Confidence>
    </Result>
    <Result>
      <ClassLabel>shadow</ClassLabel>
      <BoundingBox X="300" Y="300" Width="100" Height="40" />
      <Confidence>0.3</Confidence>
    </Result>
  </TestDataConfiguration>
</GodaConfiguration>
```

The `<GoalGeneratorConfiguration>` is consisted of two tags that are used to provide the Goal Generator with the availability of the mission’s resources - i.e. the available goals - and also the mapping between the science features and the actual goals.
Table 11-2: The <GoalGeneratorConfiguration> XML element

<table>
<thead>
<tr>
<th>XML Tag</th>
<th>Data Type</th>
<th>Parameter Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnabledGoals</td>
<td>int</td>
<td>Integer that encodes the possible goals available for the mapping. See Table 11-7 for details of the encoding.</td>
</tr>
<tr>
<td>MappingFile</td>
<td>string</td>
<td>Absolute path and name to the Goal Generator mapping file in the system, including the extension. The mapping is a .tsv/.csv file. Environment variables are allowed in the format: ${ENV_VAR}</td>
</tr>
</tbody>
</table>

The <ClassifierConfiguration> can be consisted of multiple <Model> elements in order to combine the outputs of differently trained models and extend the capabilities of the classifier. Each <Model> has a general description, a path to the actual model and a group of <Class> XML elements in the <Classes> group in order to define the available class labels used by the Classifier.

Table 11-3: The <Model> XML element of the <ClassifierConfiguration>

<table>
<thead>
<tr>
<th>XML Tag</th>
<th>Data Type</th>
<th>Parameter Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>string</td>
<td>A general name/description of the model</td>
</tr>
<tr>
<td>ModelPath</td>
<td>string</td>
<td>Absolute path and name to the model file, including the extension. The model is a .modelx file. Environment variables are allowed in the format: ${ENV_VAR}</td>
</tr>
<tr>
<td>Classes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>string</td>
<td>Name of the class as defined exactly by the model and in the correct ordering based on the output of the model.</td>
</tr>
<tr>
<td>Enabled</td>
<td>int</td>
<td>Flag to denote if the specific class will be used to generate goals. 1 means that the class is enabled and 0 that it is not.</td>
</tr>
</tbody>
</table>

The <SaliencyConfiguration> has all the required information to configure the SaliencyMap in order to be able to generalise in different environments in the case where different configurations will produce better results.

Table 11-4: The <SaliencyConfiguration> XML element

<table>
<thead>
<tr>
<th>XML Tag</th>
<th>Data Type</th>
<th>Parameter Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColorChannels</td>
<td>ColorChannel</td>
<td>Color channel transformation of the input image</td>
</tr>
<tr>
<td>ResizeToInput</td>
<td>int</td>
<td>Flag to resize the saliency map to the size of the input image. Possible values are either 1 to resize or 0 to not resize.</td>
</tr>
<tr>
<td>SubtractMean</td>
<td>int</td>
<td>Flag to normalise the input image before calculating the saliency map. Possible values are either 1 to normalise or 0 to not normalise.</td>
</tr>
<tr>
<td>GaussianBlurSigma</td>
<td>float</td>
<td>Signature output blur, expressed as fraction of the image width.</td>
</tr>
<tr>
<td>MapWidth</td>
<td>int</td>
<td>The size of the saliency map</td>
</tr>
<tr>
<td>SegmentationThresholdMin</td>
<td>float</td>
<td>The minimum value in [0,1] to consider as interesting area in a saliency map</td>
</tr>
<tr>
<td>SegmentationThresholdMax</td>
<td>float</td>
<td>The maximum value in [0,1] to consider as interesting area in a saliency map</td>
</tr>
<tr>
<td>CropXMin</td>
<td>int</td>
<td>The column of the image to start the cropping from.</td>
</tr>
<tr>
<td>CropWidth</td>
<td>int</td>
<td>The number of image columns to crop.</td>
</tr>
<tr>
<td>CropYMin</td>
<td>int</td>
<td>The row of the image to start the cropping from.</td>
</tr>
<tr>
<td>CropHeight</td>
<td>int</td>
<td>The number of image rows to crop.</td>
</tr>
</tbody>
</table>

To facilitate the integration process of the GODA to the rest of the system, artificial targets can be injected via the configuration file and the <TestDataConfiguration> element as shown in Table 11-5.
### Table 11-5: The `<TestDataConfiguration>` XML element

<table>
<thead>
<tr>
<th>XML Tag</th>
<th>Data Type</th>
<th>Parameter Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled</td>
<td>int</td>
<td>Flag to denote if the injected data should be used. 1 means that they must be used and 0 that they are not used.</td>
</tr>
<tr>
<td>Results</td>
<td>Result</td>
<td>ClassLabel: string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Name of the class as defined exactly by the classifier model.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BoundingBox: X: int, Y: int, Width: int, Height: int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Attributes to denote the top left position of the bounding box corner in (column, row) order along with the width and height of the bounding box.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Confidence: float</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The simulated confidence value of the classifier for the artificial target. Value must be in [0.0, 1.0]</td>
</tr>
</tbody>
</table>

#### 11.1.2.2. MAPPING CONFIGURATION FILE

The aim of the goal generator is to translate the detections and their associated probabilities into goals – actions defined by mission control. Therefore, the goal generator takes as input the output of the classifier and a mapping between detections and goals, and generates the actions needed to be taken based on the enabled goals by mission control and the scientific interests.

The mapping configuration file is .tsv file with the following entries as show in Table 11-6.

### Table 11-6: Description of the Goal Generator mapping file fields

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>The name of class label as defined in the configuration file. See Table 11-3.</td>
</tr>
<tr>
<td>Priority</td>
<td>Integer value in the range of [1, 10]. One (1) means is the highest scientific priority and (10) has the lowest priority.</td>
</tr>
<tr>
<td>Confidence Minimum</td>
<td>The minimum confidence required in order to consider the mapping between the detection and the goal</td>
</tr>
<tr>
<td>Confidence Maximum</td>
<td>The maximum confidence required in order to consider the mapping between the detection and the goal</td>
</tr>
<tr>
<td>Enabled Goals</td>
<td>Integer that encodes the possible goals available for the mapping. See table Table 11-7 for details of the encoding.</td>
</tr>
</tbody>
</table>

The encoding for the goal selection can be seen in the following table.

### Table 11-7: Encoding for goal selection in the mapping file of the Goal Generator

<table>
<thead>
<tr>
<th>Enabled Goals</th>
<th>KEEP_IMAGE</th>
<th>ACQUIRE_HIGH_RESOLUTION</th>
<th>REPOSITION_AND_ACQUIRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>7</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
An example of some entries of the mapping can be found in table Table 11-8 below.

<table>
<thead>
<tr>
<th>Label</th>
<th>Priority</th>
<th>Confidence Minimum</th>
<th>Confidence Maximum</th>
<th>Enabled Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outcrop</td>
<td>1</td>
<td>0.9</td>
<td>1.0</td>
<td>7</td>
</tr>
<tr>
<td>Outcrop</td>
<td>1</td>
<td>0.7</td>
<td>0.9</td>
<td>3</td>
</tr>
<tr>
<td>Outcrop</td>
<td>1</td>
<td>0.5</td>
<td>0.7</td>
<td>1</td>
</tr>
<tr>
<td>Outcrop</td>
<td>1</td>
<td>0.0</td>
<td>0.5</td>
<td>0</td>
</tr>
<tr>
<td>Float Rock</td>
<td>2</td>
<td>0.8</td>
<td>1.0</td>
<td>3</td>
</tr>
<tr>
<td>Float Rock</td>
<td>2</td>
<td>0.5</td>
<td>0.8</td>
<td>1</td>
</tr>
<tr>
<td>Float Rock</td>
<td>2</td>
<td>0.0</td>
<td>0.5</td>
<td>0</td>
</tr>
<tr>
<td>Dunes</td>
<td>8</td>
<td>0.0</td>
<td>1.0</td>
<td>1</td>
</tr>
<tr>
<td>Artificial</td>
<td>10</td>
<td>0.0</td>
<td>1.0</td>
<td>1</td>
</tr>
</tbody>
</table>

For example, let’s assume that the classifier detects an outcrop with a probability of 0.95 and a float rock with a probability of 1.0. If GODA has been called with a maximum number of goals to be 1 then it will return only the detected “Outcrop” since it has a higher priority than the “Float Rock”. If multiple “Outcrop” have been detected then they will be returned after being sorted by the confidence value and since only one is requested then only the “Outcrop” with the highest probability will be returned. Finally, the Goal Generator selects the Enabled Goals value for the given detection and decides based on the encoding what action should be made. Since in this case the mapping says “7” as enabled goals that means that all goals are available for this detection. The Goal Generator then checks the configuration file for the active goals of the mission in the current stage and selects the highest available action, with highest being the “REPOSITION_AND_REACQUIRE”, then “ACQUIRE_HIGH_RESOLUTION” and the lowest being the “KEEP_IMAGE”.

For any additional information regarding the data types and design of the GODA library please refer to the D3.1 Final Design Document.

11.1.2.3. GODA EXECUTION EXAMPLE

To facilitate the understanding of how GODA works and how it can be configured, an example is already provided. As explained in Section 8.2.1.1 the source code of the example is located in the “usage_example” folder and after a successful compilation of it, the executable will be in the “bin” folder. In order to use the example, a general configuration file, a mapping file and an image are required as described in the sections 11.1.2.1 and 11.1.2.2.

Inputs:

```
$ERGO_GODA_HOME/bin/exampleConfiguration.xml
$ERGO_GODA_HOME /bin/exampleGoalMapping.csv
$ERGO_GODA_HOME /bin/exampleImage.jpg
```
Run:

```bash
cd $ERGO_GODA_HOME/bin
./example exampleConfiguration.xml exampleImage.jpg
```

Expected Output:

```
Run:
cd $ERGO_GODA_HOME/3dGODA/bin
./example exampleConfiguration.xml exampleImage.jpg

Expected Output:

Evaluating image: /data2/repos/ERGO/GODA/bin/exampleImage.jpg
ConfigFile: /data2/repos/ERGO/GODA/bin/exampleConfiguration.xml
----- Parsing configuration file: /data2/repos/ERGO/GODA/bin/exampleConfiguration.xml
----- Resuming GODA
----- Loading models into memory:
Class Label: rock /data2/repos/ERGO/GODA/models/scisys-ml-rock_model-fold_0.modelx
Class Label: dunes /data2/repos/ERGO/GODA/models/scisys-ml-dunes_model-fold_0.modelx
Class Label: glare /data2/repos/ERGO/GODA/models/scisys-ml-glare_model-fold_0.modelx
Class Label: gravel /data2/repos/ERGO/GODA/models/scisys-ml-gravel_model-fold_0.modelx
Class Label: rocks /data2/repos/ERGO/GODA/models/scisys-ml-rocks_model-fold_0.modelx
Class Label: shadow /data2/repos/ERGO/GODA/models/scisys-ml-shadow_model-fold_0.modelx
Class Label: sky /data2/repos/ERGO/GODA/models/scisys-ml-sky_model-fold_0.modelx
Class Label: track /data2/repos/ERGO/GODA/models/scisys-ml-track_model-fold_0.modelx
Class Label: tracks /data2/repos/ERGO/GODA/models/scisys-ml-tracks_model-fold_0.modelx
----- Evaluating Image:
Image file: /data2/repos/ERGO/GODA/bin/exampleImage.jpg
Maximum goals: 10
Number of ROI found: 4
----- Generating Goals
- Generating goal for: label: rock metric: 0.409087
- Generating goal for: label: rocks metric: 0.103857
- Generating goal for: label: rock metric: 0.710616
- Generating goal for: label: rock metric: 0.272986
Number of generated goals: 4
```

Figure 11-2: Example input image
At the expected output we can see the configuration file used and the image that is under evaluation. Then we get all the information regarding the models for the classifier and finally we get the generated goals and the returned goals to the rest of the system based on the mapping file. Each goal is associated with a goal, a label, a confidence value and the region of interest in pixels. Finally, the GODA is suspended as it will no longer be used in the system since this was just an example.

If the GUI version of the library is compiled then the expected output is the same as before but images like the ones in Figure 11-3, Figure 11-4 and Figure 11-5 are also produced. These figures are the output of the Saliency algorithm, the segmented output of the Saliency Map component and the final detected bounding boxes.
11.1.3. MISSION PLANNER CONFIGURATION

11.1.3.1. REACTOR CONFIGURATION

The Mission Planner Reactor component requires a configuration file to be provided, that defines the directory and file names of the domain and problems for the use-case. It should also define the verbosity of the reactor, so that it provide differing levels of logs and initialise lambda and tick rates at start-up.

An example of a valid configuration file is as follows:

Table 11-9: Example of a Mission Planner Reactor configuration file

<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>levelAutonomy</td>
<td>E4</td>
</tr>
<tr>
<td>lambda</td>
<td>100</td>
</tr>
<tr>
<td>pi</td>
<td>1</td>
</tr>
<tr>
<td>plannerTicksPerSecond</td>
<td>1</td>
</tr>
<tr>
<td>verbosity</td>
<td>1</td>
</tr>
<tr>
<td>domain_file</td>
<td>/home/esrocos/ergo/Planner/deliberative/deliberativefunc/domains/orbital_use_case/OrbitalDomain.pddl</td>
</tr>
<tr>
<td>problem_file</td>
<td>/home/esrocos/ergo/Planner/deliberative/deliberativefunc/domains/orbital_use_case/OrbitalProblem.pddl</td>
</tr>
</tbody>
</table>

The levelAutonomy tag defines the autonomy level the reactor should be in at start-up. Setting this to E4 will allow the planner to use any provided problem with a goal to find a solution. Should this
be the case, as soon as a solution is found, the reactor will start the plan at the earliest time possible.

The lambda tag is used to define the time the planner has before a plan must be returned, expressed in ticks.

The pi tag defines the minimum upper bound on outgoing goals and observations.

Verbosity is used to set the level of logging in the reactor, with a higher value allowing a greater amount of logs to be generated.

Domain_file defines the path and file name of the selected domain file. This file is used to generate the domain data structure, check the correctness of incoming observations and goals, and is passed to the planner.

Problem_file, similarly to domain_file, selects the problem file. This file is parsed into the problem data structure, and not used again during execution. The data structure is updated during execution, and is parsed into a string for use as a parameter for the planner.

11.1.3.2. STELLAR CONFIGURATION

The Stellar planner component does not need to be configured, but must have the following points addressed:

- A folder called external_functions must be created at same level in the tree folder as the stellar_search library (.so).
- The external function libraries (.so) must be stored in a directory called external_functions which must be at the same level in the directory tree that the stellar library.
- The external functions must be defined into the domain file and the naming must be follow the next rule:
  - module_name => libmodule_name.so ....
- A folder called pddl2sas must be provided at the same level in the tree folder as the stellar_search library, or the executable created that uses this library.
- The pddl2sas python files must be stored in the pddl2sas folder.

11.2. FUNCTIONAL LAYER CONFIGURATION

The development of the functional layer for any robotic asset using ERGO's framework must address the following points:

- Identification of existing software assets and controllers that could be reused. Existing software modules must be adapted to TASTE.
- In some areas, there may not be software modules readily available, but there are existing algorithms that can be wrapped within TASTE functions, or there can be procedures comprising low-level tasks which could be reused.
- Identification of existing TASTE modules that can be used straight away. This is possible when a module addresses the control of a device that is present in the robotic platform to be controlled by the instance of ERGO under design.
- After the identification of modules to be reused or to be designed from scratch or by reusing existing procedures or algorithms, the next task is to define the corresponding views in TASTE: interface view, deployment view, data view and concurrency view.
- The constraints to be enforced at run time must be identified. These may come from mission requirements or from constraints on the robotic system. They are required to guarantee the safety of the mission.
- The suitability of the processor budget must be assessed. It shall include an estimation of the required resources, mainly CPU and memory.
The interface with the executive layer must be made explicit, including the available provided and requested interfaces. This is not only a task of the analysis of the functional layer; the requirements for this will also come from the executive.

From the TASTE generated code, the provided interfaces will need to be implemented.

### 11.2.1. GUIDANCE CONFIGURATION

The different operational modes of the Rover Guidance library (RG) are described in the final design report [AD.12].

The RG is configured by a set of (*.ini) text files. Each major component of the RG has its own configuration file. These files are located within the RG/Parameters folder, see [AD.14].

The parameter files can contain parameters necessary for the module especially describing:

- the rover configuration (e.g. size)
- the rover capability (e.g. maximal drive speed)
- the locomotion performance (e.g. slippage on sand, mean power usage)
- algorithms tuning (e.g. control gain)

The parameters names are self-explanatory. For names standard convention is used:

- names of parameters are written using **lowerCamelCase** ([https://en.wikipedia.org/wiki/Camel_case](https://en.wikipedia.org/wiki/Camel_case))
- parameters defined in particular unit has relevant postfix following the parameter name (e.g. _m for meter, _rad for radian etc.) like cellSize_m. Note: sometimes unit _pix is used however more often is omitted as non-uint value.
- parameters requiring transformation frame definition have relevant postfix (e.g. _Rb for rover body frame, _Mlg for Mars Local Geodetic frame – global reference) following name or unit postfix like fovAxisDirection_rad_Rb.

An example of parameter file (for Rover Guidance Decision Maker - RgDm) is presented in the Table 11-10 below. It contains the section name surrounded by square brackets and a list of parameters with accompanying values and comments preceeded by the `#` sign.

<table>
<thead>
<tr>
<th>#Example configuration file</th>
</tr>
</thead>
<tbody>
<tr>
<td>[RgDm]</td>
</tr>
<tr>
<td>fovAxisDirection_rad_Rb   = 0</td>
</tr>
<tr>
<td>fovWidth_rad          = 1.57</td>
</tr>
<tr>
<td>range_m_Rb           = 8</td>
</tr>
<tr>
<td>resolution_m         = 0.1</td>
</tr>
<tr>
<td>numberOfElements   = 2</td>
</tr>
</tbody>
</table>

Table 11-10 Example configuration

Please refer to the implementation for description of each parameter and understanding of how the parameters are used. Parameter files contain always full set of defined parameters and any missing parameter is showed as error message during initialisation.

Each module has a parameter definition file under:

`RG/SourceCode/GncAlgorithms/<moduleName>/DataStructDefs/<ModuleName>*_ParamStruct.h`

This should be consulted for implementation details (type, use, documentation).

### 11.2.2. ROBOTIC ARM CONFIGURATION

The configuration of the Robotic Arm is based on the tailoring of the following files:
- **RoboticArm/collision/config**: Contains all the information needed to load the robot model from urdf, the collision detection engine can also be specified. There should be one configuration file for each robot model.

- **RoboticArm/planner/config**: Specifies the path to the collision checker configuration file, the planner type and its parameters and some timeouts and ranges.

- **RoboticArm/camera/data**: Information to start the camera, to get the calibration of it and to store the marker information. There are also some parameters to improve marker detection.

- **RoboticArm/ur_cpp_driver/config**: Parameters for the UR5 arm to work properly.

- **RoboticArm/manager/data**: A configuration file for each use case with the needed parameters.

Examples of the before mentioned configuration files can be found in ERGO GitLab repository: [https://spass-git-ext.gmv.com/ERGO/RoboticArm.git](https://spass-git-ext.gmv.com/ERGO/RoboticArm.git)

### 11.2.3. FDIR CONFIGURATION

The BIP model of the FDIR component is either obtained by synthesis with the FDIR tool (the diagnoser part as the controller part is given by the user in BIP automata format) or designed by the user. In order to make such a component general enough, it is suggested that threshold or timeout parameters are made configurable. That is, they are set in a configuration file, and C++ code is written to read the file and access the values. The functions accessing the values can be used in the BIP model. An example is showed below.

```
#------------------------
# Configuration file for FDIR
#
#------------------------
#
# Time threshold for component initialization and first round execution (max allowed)
# Unit measure - milliseconds
#
max_init_time = 4000
#
# Threshold for the timeout (max allowed)
# Unit measure - milliseconds
#
wdog_timeout = 1000
#
# Epsilon for the timeout to account for delays (max allowed)
# Unit measure - milliseconds
#
wdog_eps = 0.1
#
# Battery threshold (min allowed)
#
min_battery = 50
```

(a) Configuration file

```c++
// Variable containing the configuration parameters for fdir (e.g., thresholds)
unordered_map<string, double> fdirDict;
void fdir_init_config(){
    ifstream fin("/full/path/to/config.cfg");
    for (string line ; getline(fin, line); ) {
        istringstream ssin(line);
        string name, sep;
```
double val;
ssin >> name >> sep >> val;

if (name != "" && name[0] != '#')
    fdirDict[name] = val;
}
}

const double get_init_time(){
    return fdirDict["max_init_time"];
}

const double get_wdog_timeout(){
    return fdirDict["wdog_timeout"];
}

const double get_wdog_eps(){
    return fdirDict["wdog_eps"];
}

const double get_min_battery(){
    return fdirDict["min_battery"];
}

const double get_init_time(){
    return fdirDict["max_init_time"];
}

const double get_wdog_timeout(){
    return fdirDict["wdog_timeout"];
}

const double get_wdog_eps(){
    return fdirDict["wdog_eps"];
}

const double get_min_battery(){
    return fdirDict["min_battery"];
}

(c) Using configurable thresholds in the BIP model of the FDIR component

Figure 11-6: Using configuration files for the FDIR component.

The configurable BIP FDIR model should be validated by simulation and SMC-BIP in the extended model (i.e., the BIP with faults injected). If the validation is successful, then the FDIR component can be extracted and prepared for code generation and integration within TASTE. This implies obtaining a BIP model containing only the relevant definitions with respect to the FDIR component, i.e., extern functions, port types, atom type and compound.

Depending on the properties the FDIR component enforces, changes are needed to the BIP FDIR component to be able to integrate it within a TASTE design. More specifically, if all the properties are of the type if value v is beyond threshold t, then execute p no changes are needed.
If at least one of the properties is of type \textit{if event e does not occur before timeout t, then execute p}, the input ports of the BIP FDIR model need to be changed from \textit{export} to \textit{extern}. The reason is that such properties that involve timeout events require the execution of the BIP real-time engine and therefore the interface between the environment (e.g., TASTE) and the BIP engine needs to be implemented. In the previous example the change would give:

\begin{verbatim}
extern port portBatteryPower FDIR_POWER_LEVEL_in(FDIR_POWER_LEVEL_power) as "ExternalPortBattery" error
\end{verbatim}

Where \textit{ExternalPortBattery} (given below) is the interface class between TASTE and the BIP engine. For further details about BIP the reader is referred to [RD.56].

```c++
/* Copyright VERIMAG */
#ifndef _EXTERNAL_PORT_BATTERY_
#define _EXTERNAL_PORT_BATTERY_

#include <AtomExternalPort.hpp>
#include <TimeValue.hpp>
#include <unistd.h>
#include <iostream>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <signal.h>
#include <mutex>
#include "C_ASN1_Types.h"

using namespace std;

class ExternalPortBattery : public AtomExternalPort {
public:
    ExternalPortBattery(const string &name, const EventConsumptionPolicy &policy) :
        AtomExternalPort(name, policy),
        isEvent(false) {
    }

    virtual ~ExternalPortBattery() {
    }

    virtual void initialize() {
    }

    virtual bool hasEvent() const {
        mtx.lock();
        bool r = isEvent;
        mtx.unlock();
        return r;
    }

    virtual void popEvent() {
        mtx.lock();
        assert(isEvent);
        isEvent = false;
        mtx.unlock();
    }

    virtual void purgeEvents() {
        mtx.lock();
        assert(isEvent);
        isEvent = false;
        mtx.unlock();
    }

    virtual TimeValue eventTime() const {
        mtx.lock();
        assert(isEvent);
        TimeValue r = t;
        mtx.unlock();
        return r;
    }
}
```


virtual asn1SccBatteryPower& event_get_v() {
    mtx.lock();
    assert(isEvent);
    asn1SccBatteryPower &r = val;
    mtx.unlock();
    return r;
}

void push(const asn1SccBatteryPower &v) {
    mtx.lock();
    assert(!isEvent);
    isEvent = true;
    val = v;
    t = time();
    notify();
    mtx.unlock();
}

protected:
    asn1SccBatteryPower val;
    TimeValue t;
    bool isEvent;
    mutable mutex mtx;
};

Figure 11-7: Interface and code for TASTE - BIP Engine connection of Battery port.

For this type of properties changes are needed at TASTE level too. A cyclic interface is added to the TASTE FDIR component that aims to send the BIP FDIR requests stored in a queue to other TASTE components. This is related to TASTE limitations: required interfaces can be called only from provided interfaces in TASTE. As these requests would come from the BIP Engine which runs on a different thread (not one created by TASTE for provided interfaces), they are not forwarded at execution. Then, when the BIP FDIR component executes a recovery strategy, the events and their values are pushed in queues with external code on transitions. For example,

```cpp
on FDIR_HALT_out
    from l3 to Wait
do { writelog("INFO","FDIR", "Halt","");
    wrap_halt(FDIR_HALT_haltStatus); }
(a) BIP FDIR component modifications
```

```cpp
mutex mtx;
queueOuts qo;
queueStatusOuts qso;

void wrap_halt(const asn1SccHaltStatus &val){
    mtx.lock();
    qo.push("fdir RI_HALT");
    qso.push(val);
    mtx.unlock();
}
(b) External code for the FDIR component used by TASTE template
```

Figure 11-8: Connection between BIP Engine and TASTE.

C++ code can be generated from the FDIR component with the BIP compiler. To do so, the following commands are executed.

```
$ cd /path/to/BIP/FDIR/model/and/external/code
$ source /path/to/BIP/setup.sh
$ mkdir output
$ mkdir output/build
$ cmake ..
$ make
```
The generated code is also obtained as a static library which will be used as TASTE dependency. The static library of the BIP real-time engine is also declared as a dependency in TASTE.

Finally, the interaction between TASTE and BIP model and engine is implemented by wrappers in the BIPWrapper files. To summarize, the dependencies for the TASTE and BIP generated FDIR component and BIP engine are illustrated in Figure 11-9.

![Figure 11-9: Communication and dependency diagram for a FDIR component.](image)

The BipWrapper header has the template given in Figure 11-10 and the implementation detailed below.

```c++
/* Copyright VERIMAG */
#ifndef _BIP_WRAPPER_H_
#define _BIP_WRAPPER_H_

#include <iostream>
#include <string>
#include <thread>
#include "C_ASN1_Types.h"
#include "DeployTypes.hpp"
#include "ExternalFcns.hpp"
#include <Component.hpp>
#include <AtomScheduler.hpp> //only when the BIP engine is needed

Component* deploy(int argc, char **argv);
void initialize_model();
void start_engine(); //only when the BIP engine is needed
void execute_power_level(const asn1SccBatteryPower *);
void execute_purge(); //only when BIP engine is needed
#endif
```

![Figure 11-10: Interface template for the TASTE – BIP model integration.](image)

The BipWrapper consist of:

1. Instantiation and initialization of the BIP model and BIP engine to be called from the startup TASTE template. This step is dependent on the types of properties involved.

If all properties are based on event values, then only the instantiation and initialization of the BIP FDIR model is required.

```c++
Component *fdir;
map<string, AtomExportPort *> portsFdir;

void initialize_model(){
    //Instantiate root component of bip model
    Component *root = deploy(0,NULL);
    const CT__ModelFDIR__Syst syst = *(dynamic_cast<CT__ModelFDIR__Syst*>(root));

    //Initialize fdir component from bip model
    mapstring, Component *comps = syst.components();
    auto search = comps.find("FDIR");
    if (search != comps.end())
```
fdir = search->second;
else
    writelog("ERROR","FDIR","Component not found","");
fdir->initialize();

// Retrieve ports of the fdir component
const AT__ModelFDIR__RARMxFDIR aux = *dynamic_cast<AT__ModelFDIR__RARMxFDIR*>(fdir);
portsFdir = aux.ports();
}

void fdir_startup() {
    initialize_model();
}

(b) BIPWrapper

If at least one property is based on timeout, additionally the BIP engine needs to be instantiated and executed on a thread.

Atom *fdir;
map<string, AtomExternalPort *> portsFdir;
AtomScheduler *scheduler;

void initialize_model(){
    Component *root = deploy(0,NULL);
    const CT__ModelFDIR__Syst syst = *dynamic_cast<CT__ModelFDIR__Syst*>(root);

    map<string, Component *> comps = syst.components();
    auto search = comps.find("FDIR");
    if (search != comps.end())
        fdir = dynamic_cast<Atom *>(search->second);
    else
        writelog("ERROR","FDIR","Component not found","");

    const Atom *aux = fdir;
    portsFdir = aux->externalPorts();
    // Initialize BIP engine
    scheduler = new AtomScheduler(*fdir, false);
    scheduler->initialize();
}

void start_engine(){
    // Run BIP engine
    BipError &error = scheduler->run();
}

(a) BipWrapper

(b) Taste template

The model instantiation function deploy is automatically generated by the BIP compiler in the files Deploy. These files should be copied in the TASTE template folder and merged into one file called Deploy.cc containing the static allocations from Deploy.hpp and the static allocations and functions from Deploy.cpp. Also the file DeployTypes.hpp needs to be copied in the same folder as includes the headers needed for the BIP generated component.

2. For each provided interface of the TASTE component, a wrapper is given that executes the corresponding port of the BIP FDIR component. Again, this code is dependent on the types of properties involved.

If all properties depend on the event values, then the wrapper should retrieve the corresponding port, set the value of the port, and execute it. Moreover, given the execution of the transition, the wrapper should check for all ports that are enabled by this execution and ask their execution by the FDIR BIP component. If one of this enabled ports is a recovery action, its execution should be done also in the TASTE model by calling the corresponding required interface.

void execute_power_level(const asn1SccBatteryPower *IN_powerLevel){
// Get corresponding Bip input port
AtomEPort__ModelFDIR__portBatteryPower *batPortIn =
dynamic_cast<AtomEPort__ModelFDIR__portBatteryPower*>(portsFdir["FDIR_POWER_LEVEL_in"]);

// Get port values
vector<PortValue *> pValues = batPortIn->portValues();
PV__ModelFDIR__portBatteryPower *pv =
dynamic_cast< PV__ModelFDIR__portBatteryPower*>(pValues[0]);

// Set execution value
pv->set_v(*IN_powerLevel);

// Execute port
fdir->execute("pv, TimeValue::ZERO);

// Execute enabled ports
purge();

// Method to execute all enabled Bip return ports
bool purge(){
    bool loop = true;
    while(loop){
        loop = false;
        for(auto it=portsFdir.begin(); it != portsFdir.end(); ++it){
            // Retrieve return port
            std::size_t found = (it->first).find("_return");
            if (found != std::string::npos){
                // Cast it to its type
                AtomEPort__ModelFDIR__Port *portRet = dynamic_cast<AtomEPort__ModelFDIR__Port*>(it->second);
                if(portRet->hasPortValues()){
                    // The port is enabled, execute it
                    vector<PortValue *> pValues = portRet->portValues();
                    PV__ModelFDIR__Port *pv =
dynamic_cast< PV__ModelFDIR__Port*>(pValues[0]);
                    fdir->execute("pv, TimeValue::ZERO");
                    // Loop to check for others that become enabled
                    loop = true;
                }
            }
        }
        // Retrieve recovery port (ending _out); if multiple use the name and test for each
        found = (it->first).find("_out");
        if (found != std::string::npos){
            AtomEPort__ModelFDIR__portNil *haltPortOut =
dynamic_cast< AtomEPort__ModelFDIR__portNil*>(it->second);
            if (haltPortOut->hasPortValues()){
                // Execute it in the Bip model
                pValues = haltPortOut->portValues();
                PV__ModelFDIR__portNil *pv =
dynamic_cast< PV__ModelFDIR__portNil*>(pValues[0]);
                fdir->execute("pv, TimeValue::ZERO");
                // Execute in in Taste
                fdir_RI_halt();
                // Loop to check for others that become enabled
                loop = true;
            }
        }
    }
    return loop;
}

(a) BipWrapper

void fdir_PI_POWER_LEVEL(const asn1SccBatteryPower *IN_powerLevel) {
    execute_power_level(IN_powerLevel);
}

(b) TASTE template

If at least one property is based on a timeout, then the wrapper should retrieve the corresponding port, cast it to the external port type specified in the FDIR BIP model and ask for its execution by "sending" the value to the BIP engine (via method push of the external port type implementation).

void execute_power_level (const asn1SccBatteryPower *IN_powerLevel) {
// Get corresponding Bip input port
ExternalPortBattery *batteryPortIn =
dynamic_cast<ExternalPortBattery*>(portsFdir["FDIR_POWER_LEVEL_in "]);
batteryPortIn->push(IN_powerLevel);
}

(a) BipWrapper

void fdir_PI_POWER_LEVEL (const asn1SccBatteryPower *IN_powerLevel) {
    execute_power_level(IN_powerLevel);
}

(b) TASTE template

3. If the cyclic interface is needed (when at least a property is based on a timeout), a wrapper is implemented that forwards the recovery actions to the TASTE components.

```
void execute_purge(){
    mtx.lock();
    while(qo.size() > 0){
        string sig = qo.front();
        qo.pop();
        if (sig == "fdir_RI_HALT") {
            const asn1SccHaltStatus v = qso.front();
            qso.pop();
            fdir_RI_HALT(&v);
        }
    }
    mtx.unlock();
}
```

(a) BipWrapper

```
void fdir_PI_PURGE(){
    execute_purge();
}
```

(b) TASTE template

Finally, the libraries and include dependencies are set in the user_init_pre.sh file of the TASTE model, first for the FDIR BIP model and then for the BIP engine. Then the TASTE model can be compiled and executed with the designed FDIR component included.
ANNEX A. ERGO PLANETARY USE CASE

The first use case is the planetary exploration rover, inspired from the Mars Sample Return (MSR) mission that covers the concepts and requirements of the Martian Long Range Autonomous Scientist, a Martian rover that can be commanded to operate in a multi-sol operations scheme and is able to perform long traverses. For this scenario the SherpaTT rover from DFKI was selected.

Figure 11-11: The Sherpa TT robot (courtesy DFKI)

This section describes the ERGO framework instantiation for the Planetary Exploration Demonstrator Software, illustrated in Fig. 5

Figure 11-12: ERGO framework – planetary scenario
Following the methodology described in the previous section, the architecture for the planetary rover consisted on the following components:

- As part of the agent, a Ground Control interface reactor, tailored from the generic component in the ERGO framework, handles use-case specific telemetry and telecommands. It is able to process direct telecommands (E1), time-tagged commands (E2), event-driven actions (E3) and goal commanding (E4), via the mission planner.

- High-level commands (E4) are processed by a dedicated Mission Planner reactor. This component receives high level commands that are used to generate a mission plan. This mission plan, as generated by the planner, contains a set of sub-goals to be executed at given times, together with a set of constraints to be matched. The mission planner reactor uses a specific PDDL domain and problem.

- The agent has also a scientific detector reactor. The so-called GODA reactor (that uses the GODA component provided by SCISYS) receives high-level goals in order to detect serendipitous events when the rover is traversing through specific areas by analysing images provided by the camera. When this occurs, a new goal is sent to the mission planner in order to go to a position and take images of the event detected. The planner is then able to re-plan based on the new goals.

- An additional set of reactors of the agent conform the so-called “Command Dispatcher reactors”: they interface directly to the functional layer. These receive low level goals from the mission planner (e.g. going to a desired position) and receive observations from the functional layer that indicate the results of the execution.

Finally the functional layer consists of a set of TASTE functions developed specifically for this use case, these are:

- **Antenna**: a simulated component for the antenna used to communicate with Ground.
- **Battery**: provides the battery level of the rover.
- **Camera**: interface to the Rover Cameras.
- **FDIR**: A component aimed to detect & isolate errors that could jeopardize the mission during execution.
- **Guidance Control**: Contains the Rover Guidance functionality embedded as a TASTE function. This Function uses the Guidance component detailed in SW2.
- **Planetary Robotic Arm control**: contains the Robotic arm control for the Planetary. This function uses the Robotic Arm component provided in SW2.

Following sub-sections are example of ERGO framework configuration for the planetary scenario.

### A.1. ERGO PLANETARY CASE IMPLEMENTATION

Taking into account all it has been explained in the previous sections and with the configuration files for the planetary case (Agent configuration in section A.1; and Timeline configuration in section A.3) the interface view can be modeled, as it is shown in Figure 11-13.
In the following sections both TREX components and TASTE functions are described.

A.1.1. TREX COMPONENTS – DELIBERATIVE AND EXECUTIVE LAYER

In the following tables there is a summary that shows how components described in previous sections has been instantiated for this scenario. More information about these timelines can be found in A.2 and A.3 which contains the configuration files for this case. XML is a language quite easy to read so that these files are self-explanatory at this point.

These tables link each reactor with its reactor type and corresponding internal timelines, the ones owned by each reactor.

Table 11-11: Summary of the deliberative layer TREX components for ERGO Planetary case

<table>
<thead>
<tr>
<th>Reactor Type</th>
<th>Reactor Name</th>
<th>Internal Timelines owned</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeliberativeReactor</td>
<td>PlannerReactor</td>
<td>planner</td>
<td>Planer status, whether it is planning, waiting, in an error status, etc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mission</td>
<td>Represents the high level action performed by the planner: taking sample, traversing, etc</td>
</tr>
<tr>
<td>GodaReactor</td>
<td>Goda</td>
<td>goda</td>
<td></td>
</tr>
</tbody>
</table>
Table 11-12: Summary of the executive layer TREX components for ERGO Planetary Case

<table>
<thead>
<tr>
<th>Reactor Type</th>
<th>Reactor Name</th>
<th>Internal Timelines owned</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCIReactor</td>
<td>GCI</td>
<td>- None -</td>
<td>Timeline to take and store images when requested by the planner or GCI</td>
</tr>
<tr>
<td>CMDReactor</td>
<td>CameraReactor</td>
<td>camera (AT)</td>
<td>Timeline to take and store images when requested by the planner or GCI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scientificcamera (AT)</td>
<td>Timeline to scan images (burst mode) at a given rate</td>
</tr>
<tr>
<td>CMDReactor</td>
<td>BatteryReactor</td>
<td>battery (ST)</td>
<td>State of the battery level</td>
</tr>
<tr>
<td>CMDReactor</td>
<td>AntennaReactor</td>
<td>antenna (AT)</td>
<td>Operations for the antenna component</td>
</tr>
<tr>
<td>CMDReactor</td>
<td>GuidanceReactor</td>
<td>guidancecmd (AT)</td>
<td>Operations for the guidance component</td>
</tr>
<tr>
<td></td>
<td></td>
<td>guidanceplan (ST)</td>
<td>State of the guidance planner</td>
</tr>
<tr>
<td></td>
<td></td>
<td>roverposition (ST)</td>
<td>State of the rover position</td>
</tr>
<tr>
<td>CMDReactor</td>
<td>RampReactor</td>
<td>roboticarm (AT)</td>
<td>Represent robotic arm operations</td>
</tr>
</tbody>
</table>

As explained in section 10.1.1 there is a strong bond between executive layer timelines (more precisely CMDReactors internal timelines) and TASTE interfaces, in the following table the exact correlation can be seen. Note that in all cases it follows the rules explained in section 10.1.

Table 11-13 Relation between timelines and TASTE components

<table>
<thead>
<tr>
<th>Timeline</th>
<th>Associated TASTE Interfaces</th>
<th>TASTE Data type used</th>
</tr>
</thead>
<tbody>
<tr>
<td>camera (AT)</td>
<td>CAMERA_STATUS (RI)</td>
<td>Camerastatus</td>
</tr>
<tr>
<td></td>
<td>CAMERA_REQUEST (PI)</td>
<td></td>
</tr>
<tr>
<td>scientificcamera (AT)</td>
<td>SCIENTIFICCAMERA_STATUS (RI)</td>
<td>Scientificcamerastatus</td>
</tr>
<tr>
<td></td>
<td>SCIENTIFICCAMERA_REQUEST (PI)</td>
<td></td>
</tr>
<tr>
<td>battery (ST)</td>
<td>BATTERY_STATUS (RI)</td>
<td>Batterystatus</td>
</tr>
<tr>
<td>antenna (AT)</td>
<td>ANTENNA_STATUS (RI)</td>
<td>Antennastatus</td>
</tr>
<tr>
<td></td>
<td>ANTENNA_REQUEST (PI)</td>
<td></td>
</tr>
<tr>
<td>guidancecmd (AT)</td>
<td>GUIDANCECMD_STATUS (RI)</td>
<td>Guidancecmdstatus</td>
</tr>
<tr>
<td></td>
<td>GUIDANCECMD_REQUEST</td>
<td></td>
</tr>
<tr>
<td>guidanceplan (ST)</td>
<td>GUIDANCEPLAN_STATUS (RI)</td>
<td>Guidanceplanstatus</td>
</tr>
<tr>
<td>roverposition (ST)</td>
<td>ROVERPOSITION_STATUS (RI)</td>
<td>Roverpositionstatus</td>
</tr>
<tr>
<td>roboticarm (AT)</td>
<td>ROBOTICARM_STATUS (RI)</td>
<td>Roboticarstatus</td>
</tr>
<tr>
<td></td>
<td>ROBOTICAR_REQUEST (PI)</td>
<td></td>
</tr>
</tbody>
</table>

A.2. EXAMPLE: ERGO_PLANETARY_CASE.CFG

Table 11-14: Example of Agent’s reactor configuration for the planetary scenario

```xml
<?xml version="1.0" encoding="utf-8"?>
<Agent name="ERGO" finalTick="10000">
  <!-- Normal reactors load their parameters via the TeleoReactor( xml_arg arg ) constructor. The following parameters are loaded by this constructor: -->
  - name
  - latency
  - lookahead
  - log
```
and then for each timeline we check if its external or internal.

External timelines have name, goals = 0 means no goals are posted in this timeline by this reactor. Internal timelines have name only

Plugins CMDReactor and CMDReactor are concrete reactor types which also need predicates to be defined in config file.

```xml
<Plugin name="ERGOgcireactor"/>
<Plugin name="ERGOCMDReactor"/>
<Plugin name="ERGOgodareactor"/>
<Plugin name="ERGOdeliberativeactor"/>
<Plugin name="ERGOvitre"/>

<GciReactor name="GCI"
 latency="0"
 lookahead="1"
 log="1"
 tm="1"
 tmPeriod="4"
 level="E4"
 timelinesFile="planetary_timeline_config.cfg">
    <External name="camera" />
    <External name="scientificcamera" />
    <External name="battery" />
    <External name="antenna" />
    <External name="guidancecmd" />
    <External name="guidanceplan" />
    <External name="roverposition" />
    <External name="roboticarm" />
    <External name="log" />
    <External name="planner" /> 
    <External name="mission" />
    <External name="goda" />
</GciReactor>

<GodaReactor name="GodaReactor"
 latency="0"
 lookahead="100"
 log="1"
 level="E4"
 camScanFreq="0.20"
 defaultNewGoalPriority="5"
 debugTraces="1">
    <Internal name="goda" />
    <External name="mission" />
    <External name="roverposition" goals="0" />
    <External name="scientificcamera" />
</GodaReactor>

<DeliberativeReactor name="MissionPlannerReactor"
 latency="20"
 lookahead="200"
 log="1">
    <Internal name="planner" />
    <Internal name="mission" />
    <External name="roverposition" />
    <External name="roboticarm" />
    <External name="camera" />
    <External name="antenna" />
    <External name="guidancecmd" />
</DeliberativeReactor>

<CMDReactor name="CameraReactor"/>
```
```
<cmdReactor name="battery" latency="0" lookahead="1" timeout="-1"
    timelineConfig="planetary_timeline_config.cfg">
   <Internal name="camera"/>
   <Internal name="scientificcamera"/>
</cmdReactor>

<cmdReactor name="AntennaReactor" latency="0" lookahead="1" timeout="-1"
    timelineConfig="planetary_timeline_config.cfg">
   <Internal name="antenna"/>
</cmdReactor>

<cmdReactor name="GuidanceReactor" latency="0" lookahead="1" timeout="-1"
    timelineConfig="planetary_timeline_config.cfg">
   <Internal name="guidancecmd" goals="0"/>
   <Internal name="guidanceplan" goals="0"/>
   <Internal name="roverposition" goals="0"/>
   <Internal name="log"/>
</cmdReactor>

<cmdReactor name="RampReactor" latency="0" lookahead="1" timeout="-1"
    timelineConfig="planetary_timeline_config.cfg">
   <Internal name="roboticarm" goals="0"/>
</cmdReactor>

<vitreReactor name="Vitre" latency="0" lookahead="0" log="0"
    port="31415" datadir="data">
   <External name="camera" goals="0"/>
   <External name="scientificcamera" goals="0"/>
   <External name="goda" goals="0"/>
   <External name="battery" goals="0"/>
   <External name="antenna" goals="0"/>
   <External name="guidancecmd" goals="0"/>
   <External name="guidanceplan" goals="0"/>
   <External name="roverposition" goals="0"/>
   <External name="roboticarm" goals="0"/>
   <External name="mission" goals="0"/>
   <External name="planner" goals="0"/>
</vitreReactor>
```
### A.3. EXAMPLE: PLANETARY_TIMELINE_CONFIG.CFG

#### Table 11-15: Example of Reactor's timelines configuration for the planetary scenario

```xml
<TimelineConfig>

<Timeline name="planner" tmname="TPLN" rcname="0">
  <inactivePred name="noplan" tcname="0" />
  <activePred name="loading3plan" tcname="TPLN" />
  <parameter name="file" type="string" default="needed" />
  <activePred name="planning" tcname="0" />
  <parameter name="goalList" type="string" default="needed" />
  <inactivePred name="planfailed" tcname="0" />
  <parameter name="errorCode" type="int" default="needed" />
  <activePred name="abortplan" tcname="0" />
</Timeline>

<Timeline name="mission" tmname="TMPL" rcname="0">
  <inactivePred name="idle" tcname="0" />
  <activePred name="traversing" tcname="0" />
  <parameter name="fromPose" type="point3" default="needed" />
  <parameter name="toPose" type="point3" default="needed" />
  <activePred name="takingSample" tcname="0" />
  <parameter name="toSamplePos" type="point3" default="needed" />
  <parameter name="toroverPose" type="point3" default="needed" />
  <activePred name="leavingsample" tcname="0" />
  <parameter name="toSamplePos" type="point3" default="needed" />
  <parameter name="toroverPose" type="point3" default="needed" />
  <activePred name="takingpicture" tcname="0" />
  <parameter name="toPose" type="point3" default="needed" />
  <activePred name="communicating" tcname="0" />
  <errorPred name="fault" tcname="0" />
</Timeline>

<Timeline name="log" tmname="TLOG" rcname="RLOG">
  <inactivePred name="log" tcname="CLOG" />
</Timeline>

<Timeline name="guidancecmd" tmname="TMOV" rcname="RMOV">
  <inactivePred name="idleAt" tcname="0" />
  <parameter name="atPosition" type="point3" default="lastObs" />
  <activePred name="goingto" tcname="CMOV" />
  <parameter name="fromPose" type="point3" default="lastObs" />
  <parameter name="toPose" type="point3" default="needed" />
  <activePred name="stop" tcname="CSTP" />
  <errorPred name="stuckat" tcname="0" />
  <errorPred name="at" type="point3" default="lastObs" />
</Timeline>

<Timeline name="guidanceplan" tmname="TGST" rcname="RGST" />
</TimelineConfig>
```
<inactivePred name="notinitialized" tcname="0" />
<inactivePred name="standby" tcname="0" />
<inactivePred name="plan" tcname="0" />
<inactivePred name="drive" tcname="0" />
<inactivePred name="awaitsensingcapture" tcname="0" />
<inactivePred name="awaitdemreception" tcname="0" />
<inactivePred name="safe" tcname="0" />
</Timeline>

<Timeline name="roverposition" tmname="TRPS" rcname="RRPS">
<inactivePred name="position" tcname="0"/>
<parameter name="at" type="point3" default="0"/>
</inactivePred>
<inactivePred name="unknown" tcname="0"/>
</Timeline>

<Timeline name="antenna" tmname="TANT" rcname="RANT">
<inactivePred name="idle" tcname="CAID"/>
<activePred name="communicating" tcname="CCOM"/>
<errorPred name="fault" tcname="0"/>
</Timeline>

<Timeline name="roboticarm" tmname="TPRA" rcname="RPRA">
<inactivePred name="idleat" tcname="0"/>
<parameter name="atposition" type="point3" default="lastObs"/>
</inactivePred>
<inactivePred name="picked" tcname="0"/>
<parameter name="atposition" type="point3" default="lastObs"/>
</inactivePred>
<activePred name="moving" tcname="CRAM">
<parameter name="fromposition" type="point3" default="lastObs"/>
<parameter name="toposition" type="point3" default="needed"/>
</activePred>
<activePred name="picking" tcname="CRAP">
<parameter name="sampleid" type="string" default="needed"/>
<parameter name="fromposition" type="point3" default="lastObs"/>
<parameter name="toposition" type="point3" default="needed"/>
</activePred>
<activePred name="dropping" tcname="CRAD">
<parameter name="sampleid" type="string" default="needed"/>
<parameter name="fromposition" type="point3" default="lastObs"/>
<parameter name="toposition" type="point3" default="needed"/>
</activePred>
<errorPred name="fault" tcname="0"/>
</Timeline>

<Timeline name="scientificcamera" tmname="TSCAM" rcname="RSCAM">
<inactivePred name="idle" tcname="0"/>
<activePred name="scanning" tcname="CSCAM">
<parameter name="frequency" type="float" default="1.0"/>
</activePred>
<errorPred name="fault" tcname="0"/>
</Timeline>

<Timeline name="battery" tmname="TBAT" rcname="RANT">
<inactivePred name="set" tcname="0"/>
<parameter name="level" type="int" default="lastObs"/>
</inactivePred>
</Timeline>

<Timeline name="camera" tmname="TCAM" rcname="RCAM">
<inactivePred name="idle" tcname="0"/>
<activePred name="takingpicture" tcname="CCAM">
<parameter name="filepath" type="string" default="lastObs"/>
</activePred>
<errorPred name="fault" tcname="0"/>
</Timeline>
```xml
<Timeline name="goda" tmname="TGOD" rcname="RGOD">
    <inactivePred name="idle" tcname="0" />
    <inactivePred name="goalfound" tcname="0" />
    <activePred name="searching" tcname="CGOD">
        <parameter name="conffile" type="string" default="needed" />
        <parameter name="region" type="rectangle" default="needed" />
    </activePred>
    <errorPred name="fault" tcname="0" />
</Timeline>
</TimelineConfig>
```
ANNEX B. ERGO ORBITAL USE CASE

The second scenario chosen is the On-Orbit Servicing mission, where a damaged spacecraft can have one of its modules replaced autonomously by a servicer spacecraft.

![Orbital use case](image)

**Figure 11-14: Orbital use case**

The planetary orbital demonstrator is a scenario in which a chaser spacecraft approaches a target, and is able to reconfigure it via a robotic arm. The chaser has a tray that is used to exchange a set of APMs. The chaser will reconfigure the target S/C, so the target must simulate a modular S/C with some faulty/damaged modules which the chaser will have to replace in orbit.

![ERGO orbital components](image)

**Figure 11-15: ERGO orbital components**

The architecture is depicted in Fig. 6. Note that in the orbital component, the AOCS system is not part of the ERGO components.

The Ground control interface processes E1, E2, E3 and E4 telecommands. High level goals (E4) are sent from Ground to perform changes of the configuration of the target spacecraft. The mission planner identifies the set of operations (elementary operations such as picking, or dropping an APM) to be performed by the robotic arm to reach the desired configuration(s). It does so by using specific PDDL models (domain and problem) generated specifically for this use case.

A specific set of command dispatcher reactors (battery and orbital robotic arm) interface with the functional layer to perform the corresponding commands and received the corresponding observations.
The functional layer developed for this use case includes an FDIR component, a Battery component and a robotic arm component, able to perform robotic arm path planning and execution.

Following sub-sections are example of ERGO framework configuration for the orbital scenario.

B.1. ERGO ORBITAL CASE IMPLEMENTATION

Taking into account has been explained in the previous section and with the configuration files for the orbital case (Agent configuration in section B.2; and Timeline configuration in section B.3) the interface view can be modeled, as it is shown in Figure 11-16.

![Figure 11-16 TASTE Interface view of ERGO Orbital case](image)

In the following sections both TREX components and TASTE functions are described.

B.1.1. TREX COMPONENTS – DELIBERATIVE AND EXECUTIVE LAYER

In the following tables there is a summary that shows how components described in section ANNEX B. have been instantiated for this scenario. More information about this timelines can be found in B.2 and B.3 which contains the configuration files for this case. Again the syntax and naming conventions used in these files is self-explanatory enough.

Again, these tables link each reactor with its reactor type and corresponding internal timelines, the ones owned by each reactor.

<table>
<thead>
<tr>
<th>Reactor Type</th>
<th>Reactor Name</th>
<th>Internal Timelines owned</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeliberativeReactor</td>
<td>PlannerReactor</td>
<td>planner</td>
<td>Planner status, whether it is planning, waiting, in an error status, etc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mission</td>
<td>Represents the high level action performed by the planner: taking sample, traversing, etc</td>
</tr>
</tbody>
</table>
Table 11-17: Summary of the executive layer TREX components for ERGO Orbital Case

<table>
<thead>
<tr>
<th>Reactor Type</th>
<th>Reactor Name</th>
<th>Internal Timelines owned</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCIReactor</td>
<td>GCI</td>
<td>- None -</td>
<td>- None -</td>
</tr>
<tr>
<td>CMDReactor</td>
<td>BatteryReactor</td>
<td>battery (ST)</td>
<td>State of the battery level</td>
</tr>
<tr>
<td>CMDReactor</td>
<td>RampReactor</td>
<td>roboticarm (AT)</td>
<td>Represent robotic arm operations</td>
</tr>
</tbody>
</table>

As explained in section 10.1.1 there is a strong bond between executive layer timelines (more precisely CMDReactors internal timelines) and TASTE interfaces, in the following table the exact correlation can be seen. Note that in all cases it follows the rules explained in section 10.1.

Table 11-18: Relation between timelines and TASTE components

<table>
<thead>
<tr>
<th>Timeline</th>
<th>Associated TASTE Interfaces</th>
<th>TASTE Data type used</th>
</tr>
</thead>
<tbody>
<tr>
<td>battery (ST)</td>
<td>BATTERY_STATUS (RI)</td>
<td>Batterystatus</td>
</tr>
<tr>
<td>roboticarm (AT)</td>
<td>ROBOTICARM_STATUS (RI)</td>
<td>Roboticaestatus</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ROBOTICAR_REQUEST (PI)</th>
</tr>
</thead>
</table>

B.2. EXAMPLE: ERGO_ORBITAL_CASE.CFG

Table 11-19: Example of Agent’s reactor configuration for the orbital scenario

```xml
<?xml version="1.0" encoding="utf-8"?>
<Agent name="ERGO" finalTick="10000">
  <!-- Normal reactors load their parameters via the TeleoReactor( xml_arg arg ) constructor. The following parameters are loaded by this constructor:
    - name
    - latency
    - lookahead
    - log
    - config (optional)

    and then for each timeline we check if its external or internal...
    External timelines have name, goals = 0 means no goals are posted in this timeline by this reactor.
    Internal timelines have name only

    Plugin CMDReactor are concrete reactor types which also need predicates to be defined in configfile. -->
  
  <Plugin name="ERGOgcireactor"/>
  <Plugin name="ERGOCMDReactor"/>
  <Plugin name="ERGOdeliberativereactor"/>
  <Plugin name="ERGOvitre"/>

  <GciReactor name="GCI"
    latency="0"
    lookahead="1"
    log="1"
    tm="1"
    tmPeriod="4"
    level="E4"
    timelinesFile="orbital_timeline_config.cfg">
    <External name="battery"/>
    <External name="roboticarm"/>
```
B.3. EXAMPLE: ORBITAL_TIMELINE_CONFIG.CFG

Table 11-20: Example of Reactor’s timelines configuration for the orbital scenario

```xml
<?xml version="1.0" encoding="utf-8"?>
<TimelineConfig>

<Timeline name="roboticarm" tmname="TORA" rcname="RORA">
    <inactivePred name="idleat" tcname="0">
        <parameter name="atslotid" type="slot" default="needed"/>
    </inactivePred>
    <inactivePred name="picked" tcname="0">
        <parameter name="apmid" type="apm" default="needed"/>
        <parameter name="atslotid" type="slot" default="lastObs"/>
    </inactivePred>
    <activePred name="moving" tcname="CMOVE">
        <parameter name="fromslotid" type="slot" default="lastObs"/>
    </activePred>
</Timeline>

</TimelineConfig>
```
END OF DOCUMENT