

Title **OMJava**
Java library for the OM Architectural Framework

Author Carlos Hernández

Reference R-2012-008
Release 0.4 Draft
Date March 16, 2013

Address **Autonomous Systems Laboratory**
UPM - ETS Ingenieros Industriales
José Gutierrez Abascal 2
28006 Madrid
SPAIN

OMJava

ASLab R-2012-008 v 0.4 Draft of March 16, 2013

Abstract

This document is the reference manual of the OMJava library, which is the core library of the first Java implementation of the OM Architectural Framework, known as OM1.

Keywords

Robot Control; metacontrol; machine self-awareness; model-based reflection; ROS

Acknowledgements

Table of Contents

1	Introduction	8
1.1	Metacontrolling a mobile robot	8
1.2	Overview of the OM Architecture	9
1.3	Overview of OMJava and the rest of OM1 artifacts	9
1.3.1	General remarks	12
1.3.2	Installation of OMJava	12
1.3.3	Current Status	12
1.3.4	References and documentation	12
1.4	Summary of OM1 artifacts	13
1.4.1	OM_Models	13
1.4.2	OM_Testbed1 and OM_Testbed2	13
1.4.3	OMROSDrivers_py and OMROSMetacontrol	13
1.4.4	OMJava library	13
1.4.5	org.aslab.om.ecl	16
1.4.6	org.aslab.om.metacontrol	16
2	OMJava and its application to ROS control systems	17
2.1	Introduction	17
2.2	A typical OM application	17
2.3	org.aslab.om.ros	17
2.3.1	Classes	18
2.3.2	Threads	18
2.4	org.aslab.om	19
2.4.1	Classes	19
2.4.2	Threads	20
2.4.3	Knowledge database for metacontrol	20

2.5	Overall functioning of an OM metacontrol application	22
3	Developer Manual	26
3.1	Configuring your developing environment	26
3.1.1	Example: ROS metacontroller	27
3.2	Developing the I/O between the metacontroller and your control platform	27
3.2.1	Examples	27
3.3	Creating the metacontroller's knowledge and goal	28
3.3.1	Examples	28
3.4	Putting all the pieces together	28
3.4.1	Examples	28
A	Validating Scenarios	30
B	The OM Testbeds	35
B.1	Testbed 1: Dumb control system	35
B.1.1	Rationale	35
B.2	Testbed 2: ROS simulation	37
B.2.1	Testbed 2a	39
B.2.2	Testbed 2b	41
B.2.3	Testbed 2c	46
B.3	Testbed 3: navigation of the real mobile robot	48
B.3.1	Scenario 1	48
B.3.2	Scenario 2	49

About this document

Notation

The following font styles are used to refer to special terms:

- **Concept** : any term referring to the OM Architecture
- **Artifact** : a Java project or package, or a file, that implements a piece of an OM system.
- `/ROS runtime element` : a node or a topic in a ROS system at runtime

Status

This document is still in Draft state. Pending issues are highlighted with a special format:

- Done: Scenarios defined in chapter A. Comments and approval pending.
- Done: testbed 1 has been updated. It works. Final comments and approval pending.
- Under work: testbed 2 a has been updated. It works. Final comments and approval pending.
- ToDo: review Testbeds 2b,c and 3 in the scenarios, and document here according to the format decided approved the testbed 2a.
- Review uml diagrams that document the operation of the OM Meta-controller.
- ToDo: fix usage of hyperref package

A comment concerning some pending issue for this document.

Chapter 1

Introduction

This document is the reference manual of the *OMJava* library, which is the first Java implementation of the OM Architectural Framework (called OM1). It serves two purposes: i) supply a non-exhaustive documentation of the library and how it implements the OM Architectural Framework, and ii) provide a practical manual of how to use the OMJava library.

The testbeds and scenarios that are helping develop OMJava are presented in an Annex because they provide useful examples to document OMJava.

1.1 Metacontrolling a mobile robot

At our lab, we have been developing an autonomous system testbed to drive, test, and validate our research in technology for autonomy. This testbed is a mobile robot system for navigation tasks. It consists of a differential wheel drive robot (a Pioneer 2AT), equipped with sensors such as odometry, ultra-sounds, range laser, etc., and off-board systems (computers, a tablet) to tele-operate it and execute part of the controlling system if required. Our focus is the control system.

The control system for navigation is based on that of [Marder-Eppstein et al., 2010], which uses the ROS platform [Quigley et al., 2009, Garage, 2011] and is actually the ROS official stack for navigation. We have basically adapted it to our Pioneer robot, by tuning the configuration parameters in the ROS components, and developing auxiliary components as needed.

We have also developed a simulation of the mobile robot using Stage [Gerkey et al., 2003]¹. This has supposed a change in the I/O part of the ROS control system, but the rest of it is exactly the same than for controlling the real robot. The simulation is actually another testbed, also we can better talk of a testbed family, given that it can be considered a simplified version of the same real system.

There are several issues that can challenge the autonomy of the mobile robotic system, for example:

¹<http://playerstage.sourceforge.net>

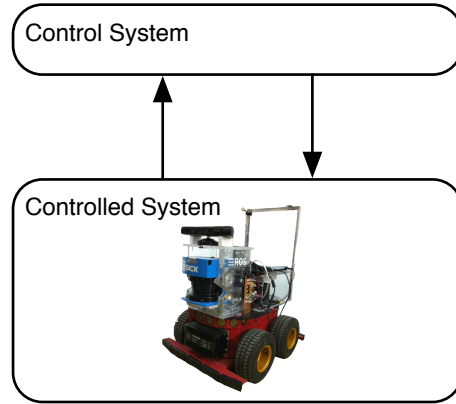


Figure 1.1: The Higgs robot and its control system.

- Run-time errors of its components: i.e. the laser stops providing readings because of an internal error.
- Wrong configuration of any of the components: i.e. the laser.
- The localization algorithm (/amcl node) not being capable of providing an estimated position

All these circumstances can cause the control system to stop working as expected, failing to provide the navigation functionality to the robot. The OM Architecture provides a solution to implement a metacontroller capable of managing (controlling) the control system by acting upon its structure, in order to maintain its functionality as far as possible.

1.2 Overview of the OM Architecture

TODO complete with text from my thesis

Refer to chapter 9 in [Hernandez, 2013] for a detailed description of the OM Architecture, and chapter 8 for more information of the metamodel it uses.

1.3 Overview of OMJava and the rest of OM1 artifacts

The first implementation of the OM Architecture has been developed as a set of software artifacts that implement the metacontroller, which is then applied for validation purposes to our testbed mobile robot. We use OM1 to refer to the complete set of artifacts developed, which includes some resources to connect the metacontroller to our testbed system. *OMJava* is the core artifact: a Java library that implements the metacontroller subsystem. Being Java, it is application and platform independent². The artifacts and the run-

²We are referring here to the platform in which the control system is implemented: i.e. CORBA, ROS, etc.

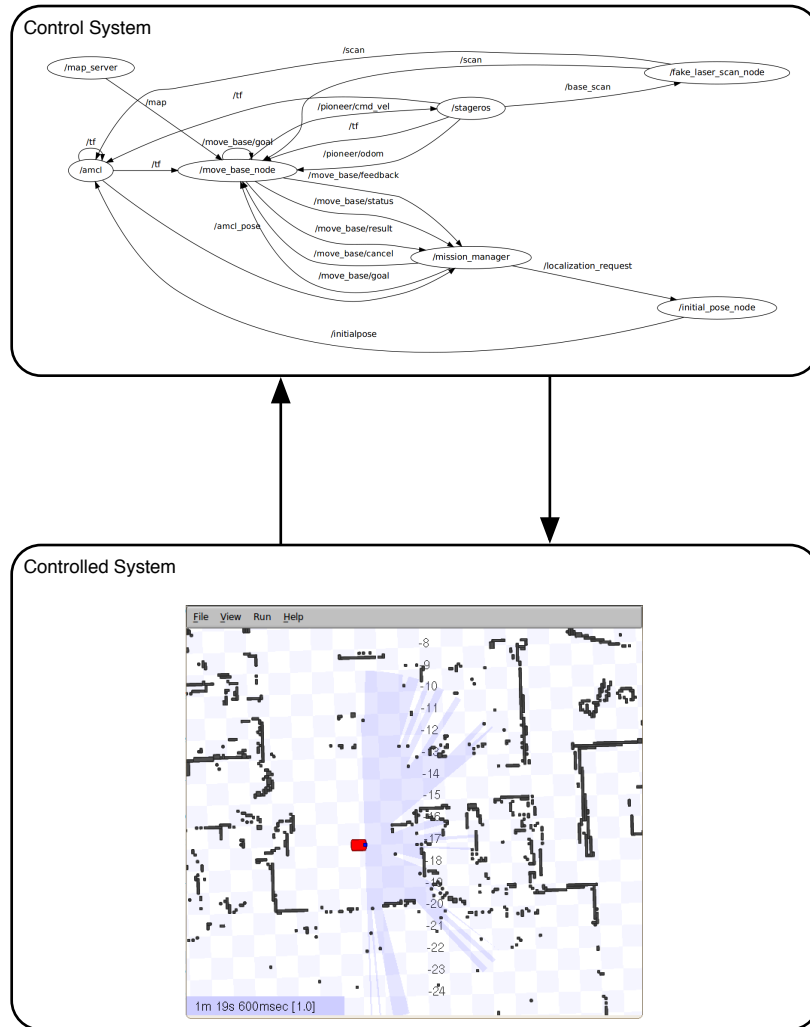


Figure 1.2: The figure shows the running simulation, with the simulated Pioneer in red, and the ROS control system for navigation, which consists of a set of running nodes (elliptical shapes) inter-connected through topics (arrows) where they publish and subscribe to messages with run-time data. For example, the `/amcl` node provides an estimation of the robot position which is published in the `/tf` topic and this way communicated to the `/move_base_node`. The `/stageros` and `/fake_laser_scan_node` nodes provide the I/O for the simulation.

time processes we have presented are illustrated in figure 1.3. This document is devoted to explain the topmost layer of the figure.

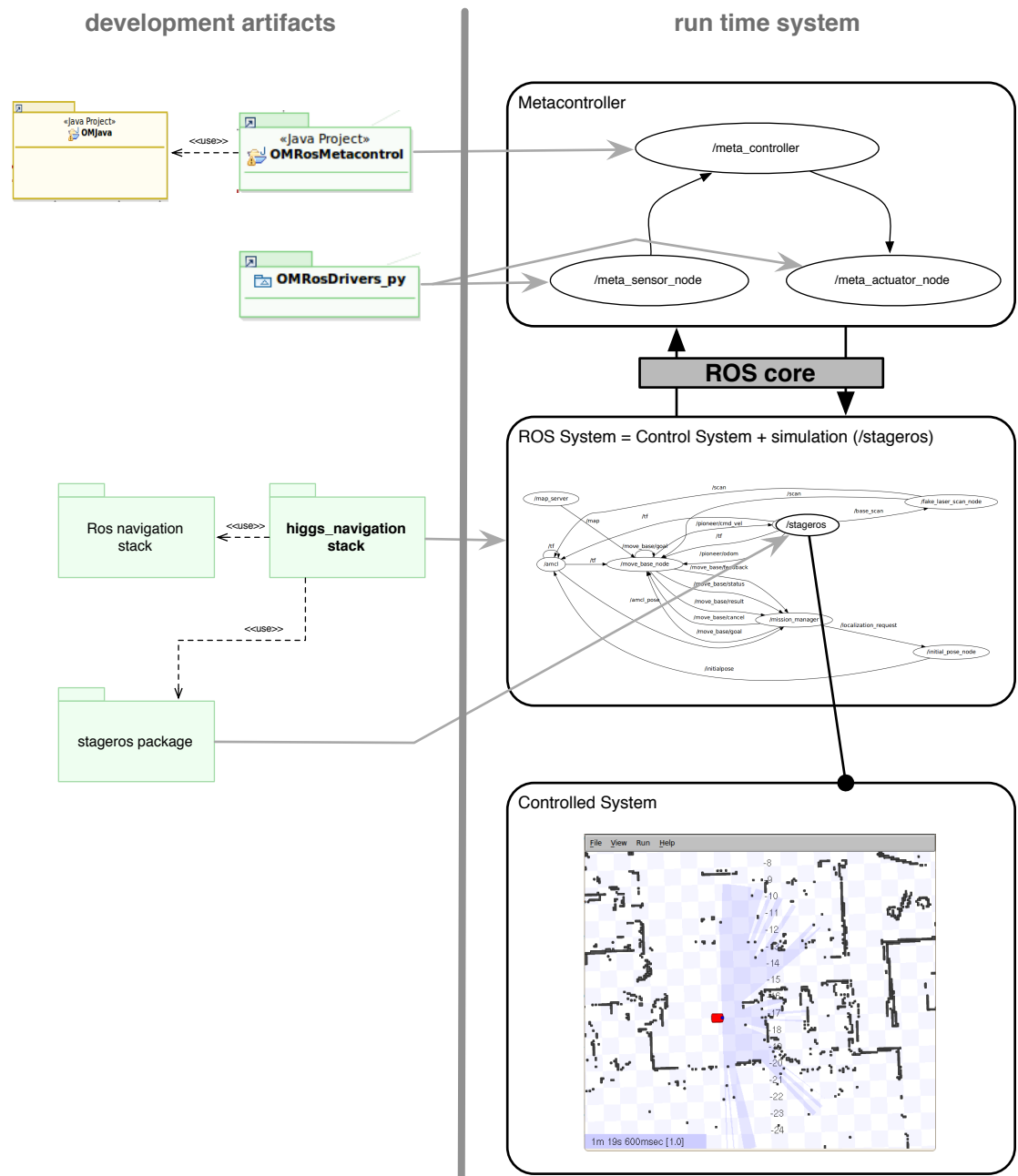


Figure 1.3: The right side of the figure depicts the system, the control system and the metacontroller, showing in each case the basic components and their connections at run time. The left side of the figure shows the OM1 artifacts to build each component. Gray arrows shows which artifact corresponds to which component. Green artifacts are ROS packages, whereas OMJava in yellow is an Eclipse Java project, whose libraries are used by OMROSMeta-control. /ROS_core refers to the ROS running infrastructure

1.3.1 General remarks

The **OMJava**³ library is still in alpha release and we only provide the sources as a Java Eclipse project. Future releases are expected to be in the form of a Java package containing the library.

1.3.2 Installation of OMJava

You can download the library from:

```
svn+ssh://software.aslab.upm.es/home/svn.repositories/Higgs/branches/ros-fuerte/OM/NamedPerception/OMJava
```

1.3.3 Current Status

Current status of development of OM1 (09/02/2012 12:35 PM) is shown in Figure 1.4 where the different Eclipse projects are shown.

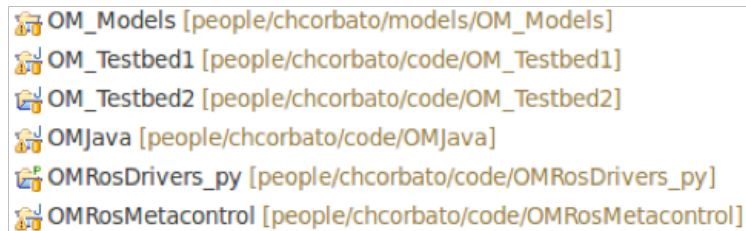


Figure 1.4: Eclipse (or RSA) projects involved in the development of OM1

1.3.4 References and documentation

Check CH thesis

for more information on all the resources related to OM1 (first implementation of The OM Architectural framework, of which *OMJava* is the keystone).

More documentation can be found in:

- OM_Models/docs (obsolete, to update)
- OMJava/doc contains the code documentation in html form produced with Javadoc.
- README.txt files and image files with diagrams in each of the Eclipse projects related to OMJava.

³for simplicity we refer to the Java library by the Eclipse project's name, although the name of its Java package is org.aslab.om

1.4 Summary of OM1 artifacts

Following is a brief overview of all the Eclipse projects used to develop OMJava (see Figure 1.5).

1.4.1 OM_Models

In this project resides most of the documentation of *OMJava*. This RSA 8 modelling project contains UML models of the artifacts resulting from the development of the *OMJava*. These models are synchronized with corresponding source code Java Eclipse projects through reconciled transformations, whose files are also stored in this project.

The model **OM.project.documentation** contains diagrams for documenting all the development related to OMJava and its testbeds. It also contains a **docs** folder with documentation, such as this document.

1.4.2 OM_Testbed1 and OM_Testbed2

These Eclipse projects contain the sources for each of the testbeds envisioned in the development of *OMJava* (see (Hernandez, 2012) and B).

1.4.3 OMROSDrivers.py and OMROSMetacontrol

These two Eclipse projects are also ROS packages, containing the infrastructure to connect the metacontrol system implemented in the OMJava library to a ROS-based system⁴.

The project **OMRosDrivers.py** contains the implementation of ROS nodes in Python for the meta actuator (**meta_actuator_node.py**) and meta sensor (**meta_sensor_node.py**) used by the metacontroller to perform **meta-control** of a running ROS system. They can be run also independently of the metacontroller. Testing applications, consisting of python executables which are also ROS nodes, are included as examples.

OMRosMetacontrol contains the implementation of a ROS Java node for interfacing the *OMJava* library with ROS, named `/meta_controller`. The main class **OMMetacontroller**, wraps an instance of `org.aslab.om.metacontrol`. It senses through the ROS topic published by the **meta_sensor_node.py**, and acts through the ROS topic `/meta_action` of the ROS node **meta_actuator_node.py** (see figure below).

1.4.4 OMJava library

The library is organized in Java packages (see Figure 1.7) according to the structure of the OM Architectural Framework conceptual described.

⁴<http://www.ros.org/>

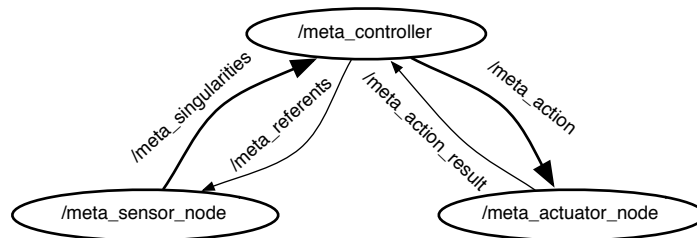


Figure 1.6: Basic ROS nodes of the meta interface.

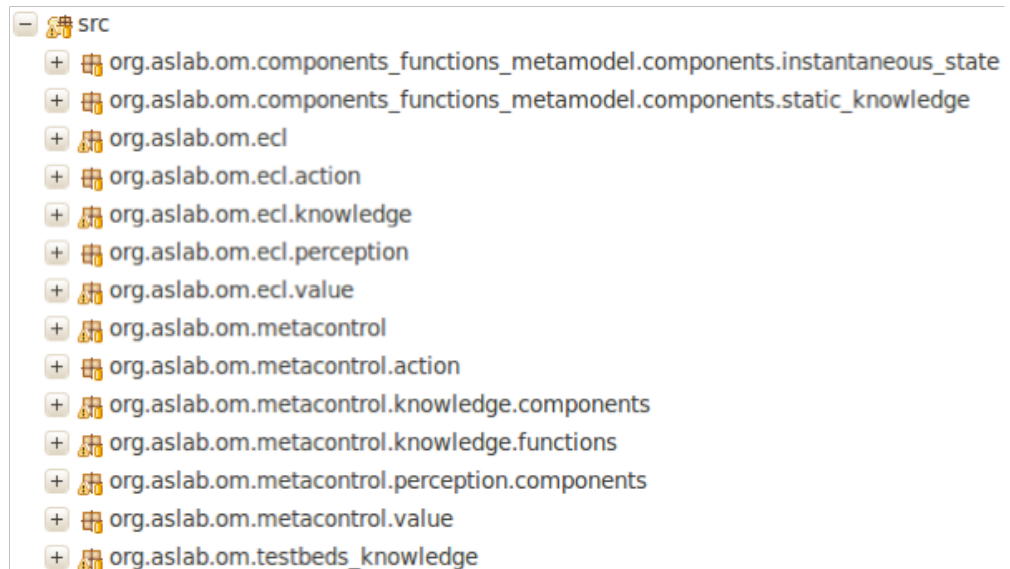


Figure 1.7: Exhaustive list of all the Java packages that conform OMJava library.

1.4.5 org.aslab.om.ecl

This package provides a Java implementation of the Epistemic Control Loop Architectural Framework, each of its subpackages dedicated to a specific aspect of the ECL: action, knowledge (models), value (perception is still to be done).

The keystone of the package is the **ECL** class, which defines the model-based control loop. Its *loop()* method is called periodically from the class **thread**.

1.4.6 org.aslab.om.metacontrol

This package contains classes to realise a **metacontrol** according to The OM Framework. For that it uses and refines elements from the *ecl* package.

The *OMmetacontroller* class implements a metacontroller from a given initial knowledge and goal, and provides basic managing of its execution. An OM-metacontroller instance manages the execution of the threads of a **ComponentsECL** and **FunctionaleECL** objects, which implement the nested ECL loops defined in the OM Architectural Framework.

The **PlantAPI** abstract class defines the interface of an **OMmetacontroller** instance with the plant it (meta-)controls.

Describe other packages?

Chapter 2

OMJava and its application to ROS control systems

2.1 Introduction

This chapter provides an overview of a typical OM application, exemplified with Testbed2. The section is organised according to the Java packages involved, describing for each one the principal classes and how they contribute to the application.

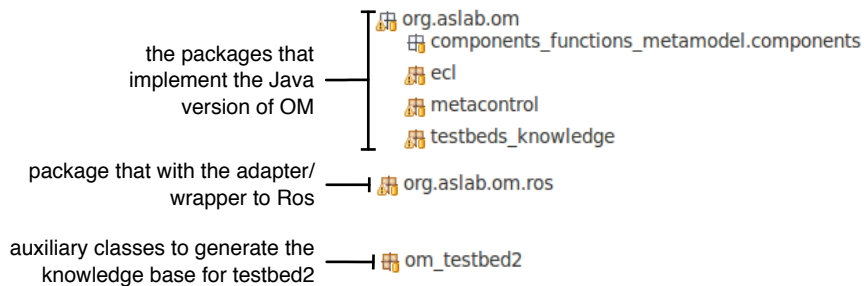


Figure 2.1: The Java packages involved in the development of OM1 and the testbeds (do not confuse with the Eclipse projects, although each one of the top level packages is in each one of the core Eclipse projects for OM1)

2.2 A typical OM application

Add here a brief description of an example of an OM application e.g. Testbed2 ?

2.3 org.aslab.om.ros

This package, in the **OMROSMetacontrol1** Eclipse project, contains all the classes needed to instantiate an OM **metacontroller** in a ROS system. It

requires the nodes in **OMROSDrivers.py**, since it subscribes and publishes to that topics for meta I/O.

2.3.1 Classes

OMMetacontrollerNode : contains the *main()* thread, where it:

1. Creates an instance of **org.ros.node.Node** to connect to the ROS master (i.e. the “broker”). Documentation about this class can be found in documentation of the rosjava stack at www.ros.org, although it is extremely unstable, the version used in *OMJava* dating of September 2011 and being already deprecated.
2. Creates an instance of **OMmetacontroller**, that is the OM **metacontrol** system.
3. Creates an instance of **OMRosAPI** and connect the metacontrol to it for meta I/O.

OMRosAPI : contains subscribers to the meta Input and publishers to the meta Output. The subscribers create asynchronous threads upon message reception to handle them

Figure 2.2 depicts the main classes and the flows of data.

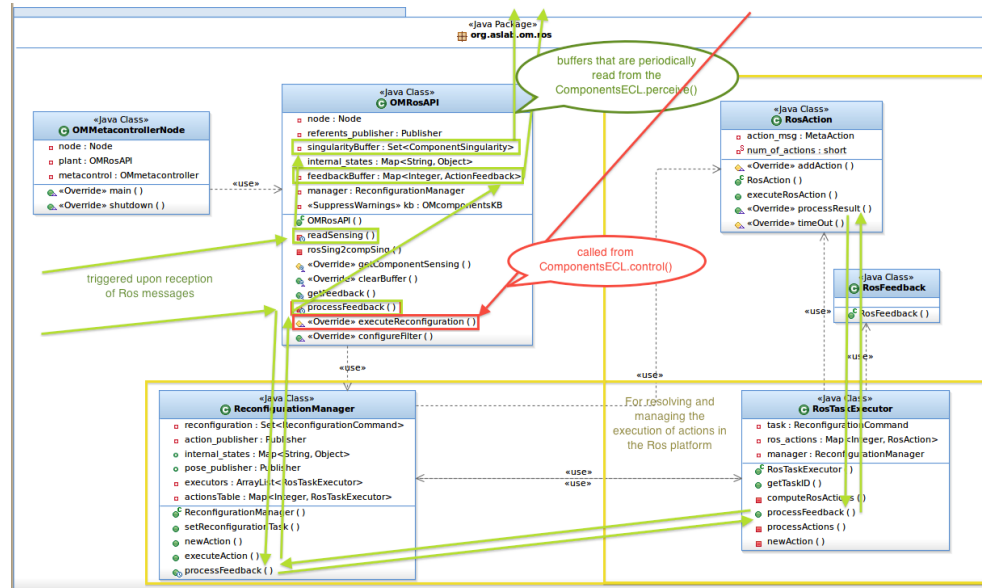


Figure 2.2: Main classes and flows of data.

2.3.2 Threads

There is one asynchronous thread created by the ROS infrastructure to attend to each incoming message, either in the `/meta-singularities` and the

/meta_action_result topics:

- A /PerceptionFlow message incoming in the /meta_singularities topic, is processed to convert it to **Set<ComponentSingularities>** and then it is put in the **singularityBuffer** (which is previously emptied)
- A /MetaActionResult message incoming in the /meta_action_result topic is processed by **OMRosAPI.manager** resulting in the update the state of its internal **ROSActions** and **ROSTaskExecutors**. This can result in turn in either:
 - a new **ROSAction** may be commanded by publishing the corresponding **MetaAction** message in the /meta_action topic)
 - a feedback on the task (**ReconfigurationCommand** instance) of which the **ROSAction** pertains, which is put in the **feedbackBuffer**.

2.4 org.aslab.om

The Eclipse project OMJava contains the org.aslab.om package and subpackages. The two core subpackages are:

org.aslab.om.ecl: contains an implementation of the Epistemic Control Loop Pattern.

org.aslab.om.metacontrol: contains the implementation of the OM Architectural Framework, using elements in the previous ***.ecl** package

2.4.1 Classes

***.ecl.ECL** defines the basic control loop as a periodic execution:

$$perceive() \rightarrow evaluate() \rightarrow control()$$

***.metacontrol.FunctionalECL** contains everything for the functional level of OM

***.metacontrol.ComponentsECL** contains evaluation and control procedures for components, but the perception is delegated into:

***.metacontrol.perception.ComponentsPerceptor**

2.4.2 Threads

There are two threads, corresponding to the control loops at the components and functional layers as defined by the Functional Metacontrol pattern. They are therefore owned by the **FunctionaleECL** and **ComponentsECL** classes respectively. Both threads execute `org.aslab.om.ecl.ECL.run()`, which consists of a periodic execution of `org.aslab.om.ecl.ECL.loop` (`perceive()` -> `evaluate()` -> `control()`).

2.4.3 Knowledge database for metacontrol

The knowledge that the metacontrol uses about components and functions consists of a set of objects compliant with our metamodel for functions and components. There are three classes that provide different ways of accessing these objects, as shown in the following figure:

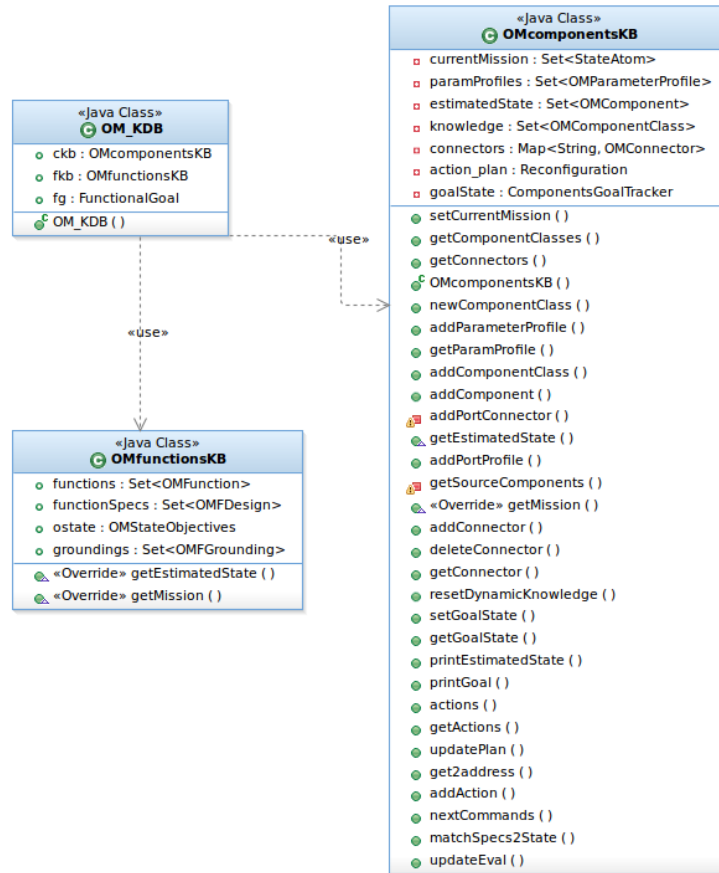


Figure 2.3: Classes to get access to OM metamodel-compliant classifiers.

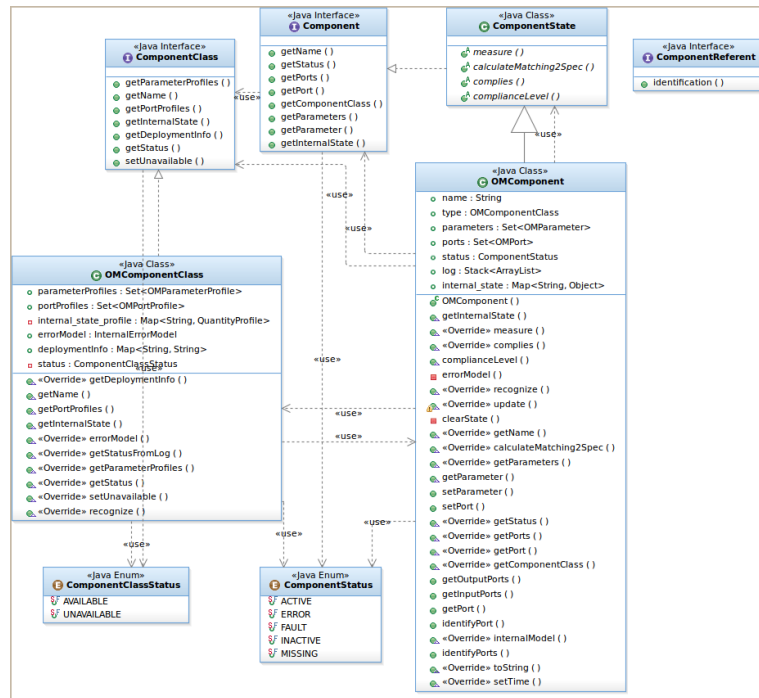


Figure 2.4: Classes to implement knowledge about components.

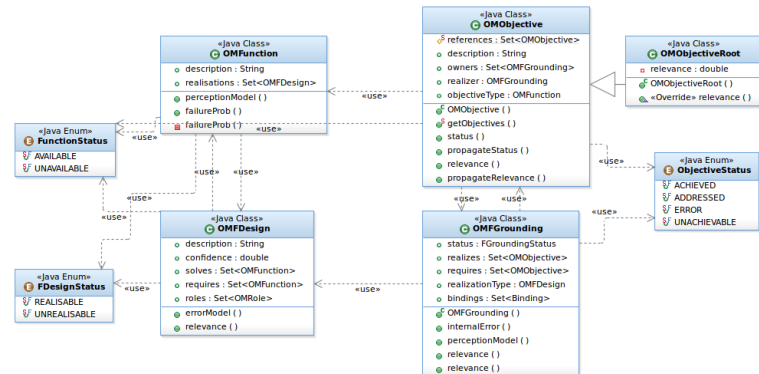


Figure 2.5: Classes to implement knowledge about functions.

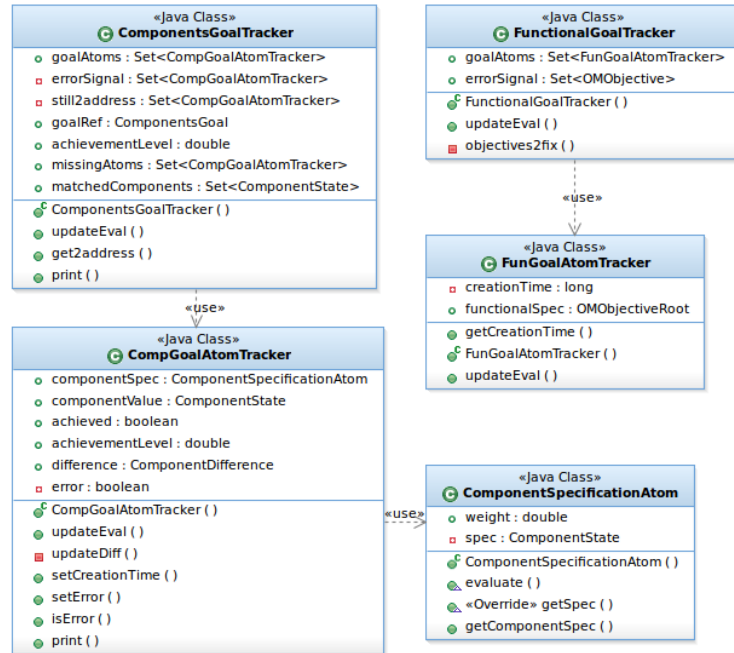


Figure 2.6: Classes to define goals.

2.5 Overall functioning of an OM metacontrol application

Add something about the meta interface?

Describe OMRosMetacontrol?

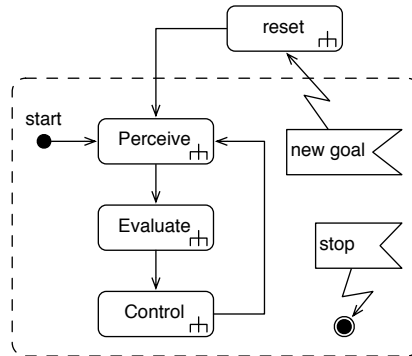


Figure 2.7: The activity diagram of the the ECL class, which is the one that follows the FunctionECL and the ComponentsECL

See also pages 48 and 49 for activity diagrams of the OM metacontroller corresponding to scenarios 1 (transient laser failure) and 2 (permanent laser failure) in Testbed3 (the real mobile robot Higgs).

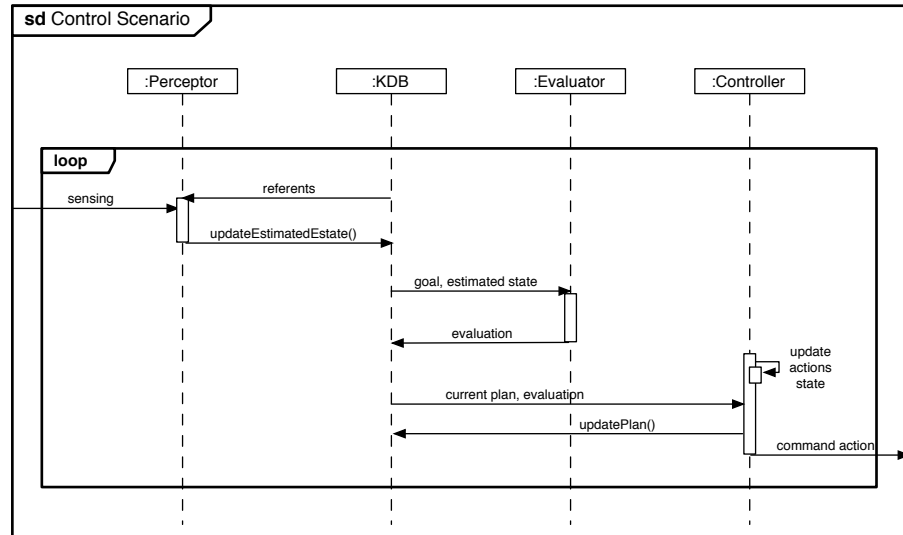


Figure 2.8: The sequence of messages in the standard scenario between the components of the ComponentsECL.

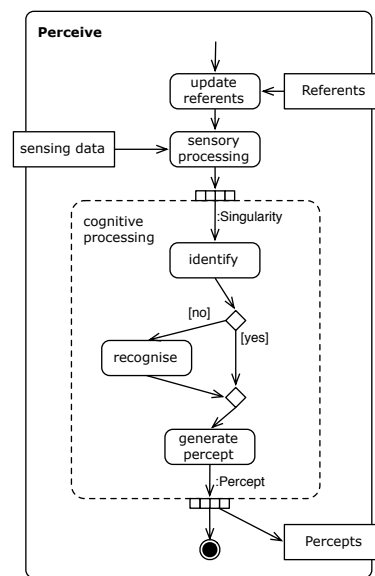


Figure 2.9: The activity diagram of the ExplicitPerceptor class, which is the one defined for the ComponentPerceptor

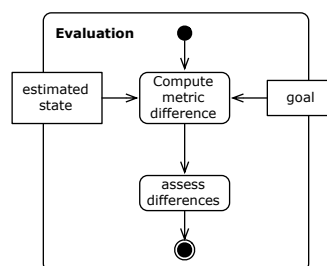


Figure 2.10: The activity diagram of the ComponentsECL.evaluation()

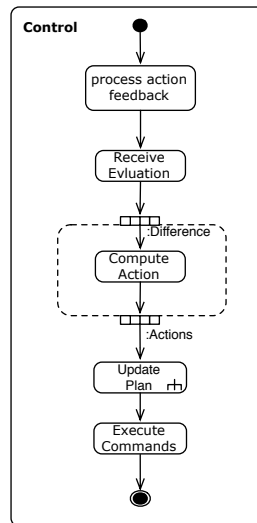


Figure 2.11: The activity diagram of the `ComponentsECL.evaluation()`

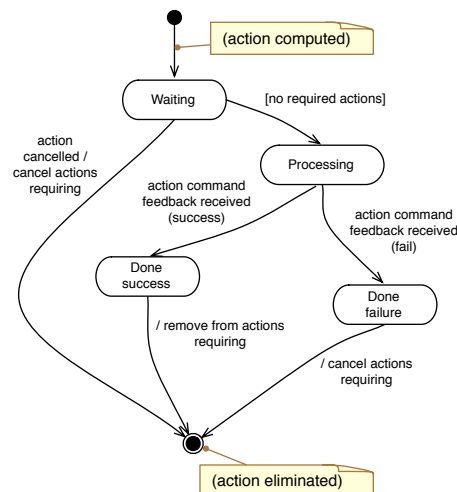


Figure 2.12: State machine of `ComponentsAction`

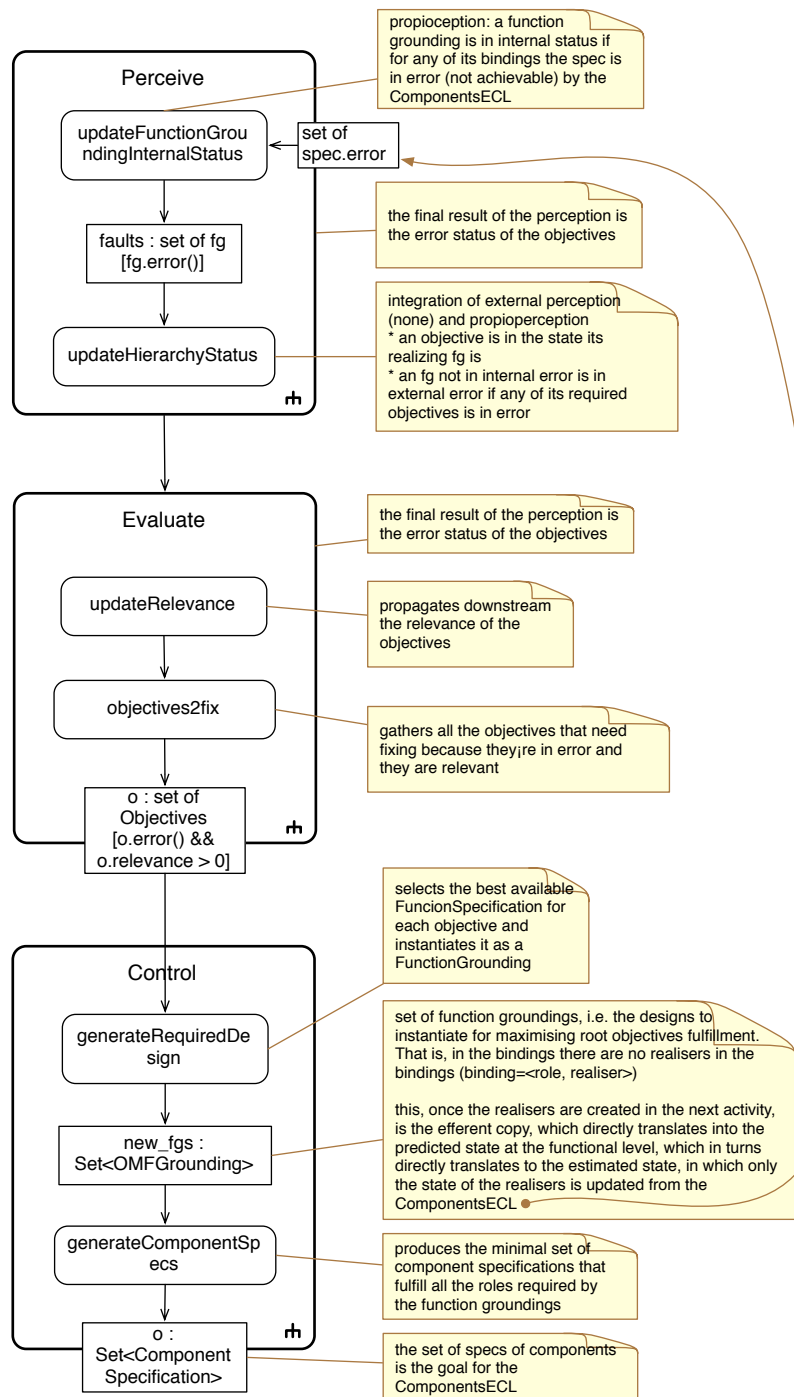


Figure 2.13: Activity diagram for the FunctionalECL class

Chapter 3

Developer Manual

How to use the OMJava library to develop a metacontroller for your application

This section is a brief manual on how to implement a metacontroller according to the OM Architectural Framework by using *OMJava*.

This manual supposes basic knowledge of the Eclipse platform for Java development. The metacontroller will be implemented as a Java application to run in a JavaSE-1.6 environment, although it may work on other Java environments.

The system to metacontrol (plant) can be in any platform, as far as an interface is provided to connect to the metacontrol. Check the projects for an example on how that was done for ROS.

The main steps for incorporating a OM metacontroller to your control system are:

1. Configure your development environment.
2. Develop a connector for the I/O between the metacontroller and the control system platform.
3. Create the initial knowledge for the metacontroller.
4. Put all the pieces together

The following sections detail the actions required in each of these steps.

3.1 Configuring your developing environment

This manual considers you are using Eclipse for developing your metacontroller. The initial configuration before you start developing your metacontroller is quite simple:

1. Import the **OMJava** project into your Eclipse workspace. For example, you can use a SVN plug-in, such as Subclipse or Subversive, to check-it

out from the ASLab SVN repository into your workspace.

Repository URL:

svn+ssh://software.aslab.upm.es/home/svnroot/people/chcorbato/code/OMJava

2. Build the project OMJava
3. Now you can create a Java project for your metacontroller, e.g. **MyMetacontroller**. Configure the following properties for the project:
 - Add **OMJava** to its referenced projects.
 - Add any dependency you will require to develop code to connect to your control platform.

Following

3.1.1 Example: ROS metacontroller

We have already developed a metacontroller for control systems implemented with the ROS platform, in a project called **OMRosMetacontroller** (which would correspond to **MyMetacontroller**). In the case of ROS it is simpler to create the eclipse project using the ROS build system, since it generates for us all the project configuration required for the ROS platform, such as library dependencies. This process consists of creating a new ROS Java package, then generating the eclipse configuration files and finally importing the project into our Eclipse workspace.¹

You can check out If your control system is ROS you can simply check-out the **OMRosMetacontroller** project into your ROS workspace. To work with it make sure to also add it to your `ROS_PACKAGE_PATH` (ros-electric: check-it out in a directory that is already there, e.g. the directory you have for your ROS projects, ros-fuerte: use `rosws set`).

3.2 Developing the I/O between the metacontroller and your control platform

You have to extend the `org.aslab.om.ecl.PlantAPI` abstract class to implement the I/O of the metacontroller with your platform. Following with provide two examples of how this can be done.

3.2.1 Examples

Dumb tester

For testbed 1: `om.testbed1.FakedROSDriver` implements a dumb interface: no sensing is provided and actions only result in stdout printing.

¹see www.ros.org for details

ROS I/O

For the ROS platform (e.g. testbeds 2, 3 and 4) the I/O developed has been described in section 1.4.3. The `org.aslab.om.ros` package contains classes that implement the connection to the ROS platform. The core class is `OMRosAPI`, which implements the `PlantAPI` interface

3.3 Creating the metacontroller's knowledge and goal

The final step to create your metacontroller is building the initial knowledge for it. You have to create an instance of `org.aslab.om.metacontrol.knowledge.OMKDB` and populate it with instances of the classes in the `org.aslab.om.metacontrol.knowledge` package according to the functions and components in your control system package.

3.3.1 Examples

Knowledge about typical ROS components is implemented by the classes in the Java package:

`org.aslab.om.testbeds.knowledge`

To initialize the metacontroller knowledge (control system specific knowledge, goal and initial state) the following classes can be used, which programmatically construct every element of the **Knowledge Database**.

For testbed 1: `om_testbed1.Testbed1_KDB`

For testbeds 2a, 2b and 2c: `om_testbed2.Testbed2x_KDB`

3.4 Putting all the pieces together

Implement your application main class that instantiates an `OMmetacontroller` instance with the initial knowledge and goal, and uses its public methods to control its execution.

3.4.1 Examples

For testbed 1: `om_testbed1.FreeEvolutionTest`

ROS Metacontroller

For ROS systems `org.aslab.om.ros.OMMetacontrollerNode` wraps everything into a ROS node coded in Java by using the rojava ROS stack. `OMMetacontrollerNode` creates and connects together as appropriatedly

instances of **OMmetacontroller**, **OM_KDB**, a ROS node and **OMRosAPI**, so that the instance of **OMmetacontroller** (the metacontroller) is connected to the ROS-based control system (via the **org.ros.Node** instance) through the **OMRosAPI** methods and buffers.

The initial knowledge is loaded by instantiating one of the **om.testbed2.Testbed2x_KDB** classes, according to the `/testbed` ROS parameter, which can be configured in the ROS launch file for the system. See **OM_Testbed2/launch/testbed2_b.launch** for an example.

Appendix A

Validating Scenarios

The goal of the metacontroller defined in OMJava is to manage the runtime control system so that it keeps achieving the system's objectives. The following general scenarios were considered in the design of the OM Architecture (see CH thesis

, The metacontrol scenarios):

Scenario 1 One or several components of the control system undergo a transient failure. The metacontroller is expected to recover those components from the failures, maintaining the configuration of components. This scenario, when the system actual configuration is incorrect, can also be due to an erroneous initialisation or whatever) *I have this as a different scenario in the thesis*. The metacontroller fixes the configuration.

Scenario 2 One or several components undergo a permanent failure and no components that could perform their roles are available, that is: the desired configuration of the system is not feasible. The metacontroller re-design the system and reconfigures it accordingly.

Scenario 3 The control system fails to achieve its functional requirements due to an unknown cause *explain*. The control system is re-designed and reconfigured accordingly.

the 4 scenarios can be grouped in two: component recovery or actual re-design

To validate the implementation of OMJava, the following operational scenarios for our testbed family have been considered (from CH thesis

):

1. S0 – No failure in patrolling system: the Metacontrol receives monitoring information from the patrolling system and maintains a representation of its state. The Metacontrol does not take any action because the patrolling system is working as desired.
2. S1 – Laser transient failure: during regular operation, the robot's laser driver fails and no laser information is received by the patrolling system. The Metacontrol observes this and re-starts the laser's driver.

3. S2 – Laser permanent failure: same incident with the laser's driver than above, but this time re-starting the driver does not solve the problem. The Metacontrol reconfigures the patrolling system so as to use the kinect sensor input data instead of the laser's for navigation (kinect configuration).
4. S3 – Localisation error while in kinect configuration: the patrolling system malfunctions while in the kinect configuration because of the localisation module not being able to estimate a robot position. The Metacontrol detects this and reconfigures the system in the laser configuration.

Following these scenarios are detailed:

Scenario 0	No failure in patrolling system
Description	the Metacontrol receives monitoring information from the patrolling system and maintains a representation of its state. The Metacontrol does not take any action because the patrolling system is working as desired.
Preconditions	the patrolling system is up and running, the metacontroller system is also running.
Triggering event	the patrolling system is working properly with the configuration desired.
Basic Flow	the metacontroller periodically updates its structural model of the controlling system, by updating the state of the components. Since that state complies with the desired design, no action is needed.
Postconditions	the patrolling system works as desired without interference from the Metacontrol

Scenario 1	Laser transient failure
Description	during regular operation, the robot's laser driver fails and no laser information is received by the patrolling system. The Metacontroller observes this and re-starts the laser's driver.
Preconditions	the patrolling system is up and running in laser configuration, the metacontroller system is also running.
Triggering event	the laser driver fails and stops providing laser data to the patrolling system.
Basic Flow	<ol style="list-style-type: none"> 1. detect errors in the components of the patrolling system. 2. update the state of components observed: the laser driver is in error. 3. re-start the laser driver.
Postconditions	the patrolling system has resumed its mission by navigating to the waypoint it targeted when this scenario triggered, operating with the laser configuration as before the triggering event. The Metacontrol is also up and running as previously, monitoring the state of the patrolling system.

Scenario 2	Laser permanent failure
Description	the laser driver stops working and re-starting it does not solve the problem. The Metacontrol reconfigures the patrolling system so as to use the kinect sensor input data instead of the laser's for navigation (kinect configuration).
Preconditions	the patrolling system is in laser configuration, but not running because of a laser driver failure, which has caused the system to undergo the laser transient failure scenario, not having returned to the normal operation scenario.
Triggering event	No laser component can be instantiated successfully.
Basic Flow	<ol style="list-style-type: none"> 1. detect errors in the components of the patrolling system. 2. update the state of components observed: several functions are in failure, but the origin is an internal failure of the function to provide laser readings. 3. update the functional status of the system: the function to obtain laser readings is no longer available. 4. reconfigure the patrolling system in kinect-only mode.
Postconditions	the patrolling system has resumed its mission by navigating to the waypoint it targeted when this scenario triggered, but now in the kinect-only configuration. The Metacontrol is also up and running as previously, monitoring the state of the patrolling system.
Includes	the laser transient failure scenario, but for its postconditions, which do not hold.

Scenario 3	Kinect localisation failure
Description	during operation in kinect-only mode, the localisation component stops producing an estimation of the robot position. The Metacontrol observes this and reconfigures to use laser readings for localisation and navigation.
Preconditions	the patrolling system is up and running in kinect-only configuration, the Metacontrol system is also running.
Triggering event	the localisation component gives a warning message and stops publishing an estimated position for the robot.
Basic Flow	<ol style="list-style-type: none"> 1. detect errors in the components of the patrolling system. 2. update the state of components observed: the laser driver is in error. 3. start the laser driver and reconfigure localisation and navigation components to use its published scans.
Postconditions	the patrolling system resumes its mission by navigating to the waypoint it targeted when the scenario triggered, but now operating with the mixed configuration. The Metacontrol is also up and running as previously, monitoring the state of the patrolling system.

Appendix B

The OM Testbeds

Testing applications for the Operative Mind Architectural Framework

This chapter describes the testbed applications that are being developed to demonstrate the validity of The Operative Mind Architectural Framework (OM1) to develop self-awareness mechanisms to robustly improve the autonomy of control systems. This work is part of Carlos Hernández's PhD thesis. These testbeds are proved in the defined scenarios to validate their fault-tolerance capacities.

B.1 Testbed 1: Dumb control system

Purpose: Minimal validation (internal consistency) of the implementation of the OMJava library.

B.1.1 Rationale

The aim of this testbed is to validate the internal consistency of the implementation of OM1. It consists of a metacontroller running standalone, not connected to any control system but to a dumb testing process.

The testing process implements a dumb MetaI/O interface, which provides no monitoring information to the metacontroller, and processes the reconfiguration actions received by simply printing them in the standard output.

The metacontroller activity departs from a given initial knowledge about the state of the control system (both about available functions and components, and an initial state in terms of grounded functions and components' configuration) and a goal, and will generate appropriate action.

The metacontroller knowledge consists of a component type (laser) with a parameter profile (baudrate), and a function to have laser readings and a

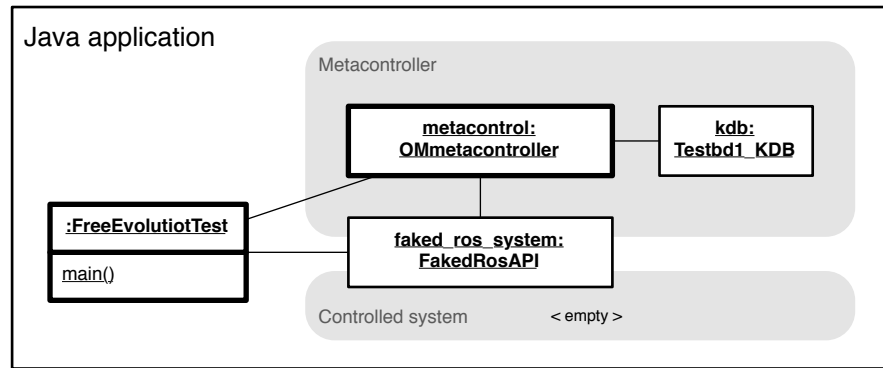


Figure B.1: The basic objects at runtime in the Dumb Control System testbed. The FreeEvolutionTest class is the main class of the Java application. It creates and connects an OMmetacontroller object and an instance of FakedRosAPI, which is a dumb simulation of a control system implementing the meta I/O interface. The kdb instance of Testbed1_KDB implements the initial knowledge of the metacontroller.

function design to achieve it. This knowledge is hand coded.

In the application, an instance of the metacontrol is created with that knowledge, the goal to maintain the root objective to have laser reading, and the initial state an instance of the laser component with a value for its baudrate of 19200 (different from that specified in the function design). No sensing input is given, and actions are sent from the metacontrol to launch the laser component. The actions have the only result of having them printed onscreen, since the dumb I/O interface only does so.

Expected result: the code executes without error, and the action to reconfigure the laser component is output.

Developed artifacts:

- *OM.Models*¹ RSA project model
 - *OM1ArchitecturalFramework* model (with *ECL* and *metacontroller* packages)
 - *OM.Testbed1* model
- Code:
 - *OMJava* Eclipse project
 - *OM.Testbed1* Java Eclipse project

Due date: January 29 — Status: done — updated to work with new OMJava version (July 18 2012)

¹Artifacts in italics are common to all testbeds and are a final deliverable

B.2 Testbed 2: ROS simulation

Testbed 2 consists in the application of the OM metacontroller to the ROS-based control system of a simulation of our mobile robot. It is decomposed into a family of testbeds 2a, 2b, and 2c, that share a common implementation of OM for a ROS control application.

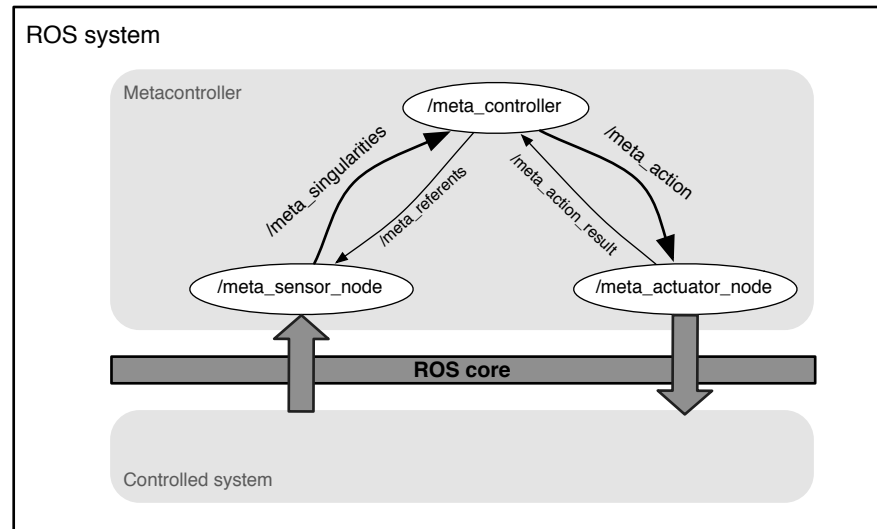


Figure B.2: The metacontrol system implement for Testbed2. It is the same for testbed3 (the real robot).

Each of the testbeds a, b, c correspond to a different control system, with an increasing degree of complexity:

Testbed 2a: the simulated control system consists of a single component, the `/fake_laser_scan_node`, which simulates the laser driver.

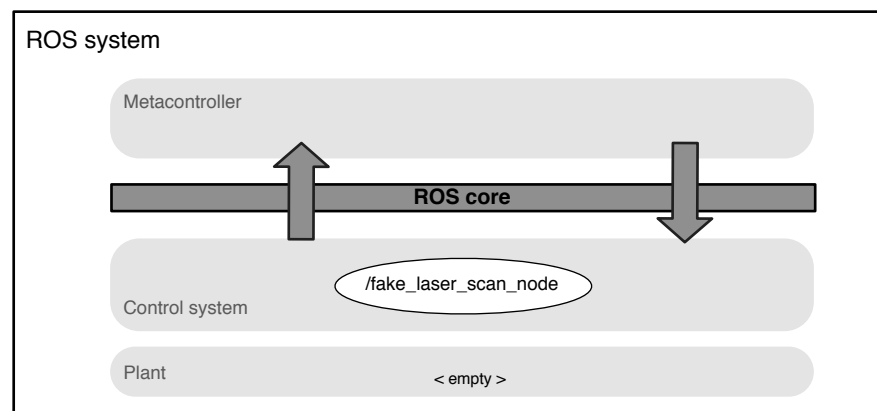


Figure B.3: The testbed 2a

Testbed 2b: the simulated control system includes the robot simulation, using the `/stageros` node from the ROS stack, and the `/fake_laser_scan_node`

connected to it to provide laser readings.

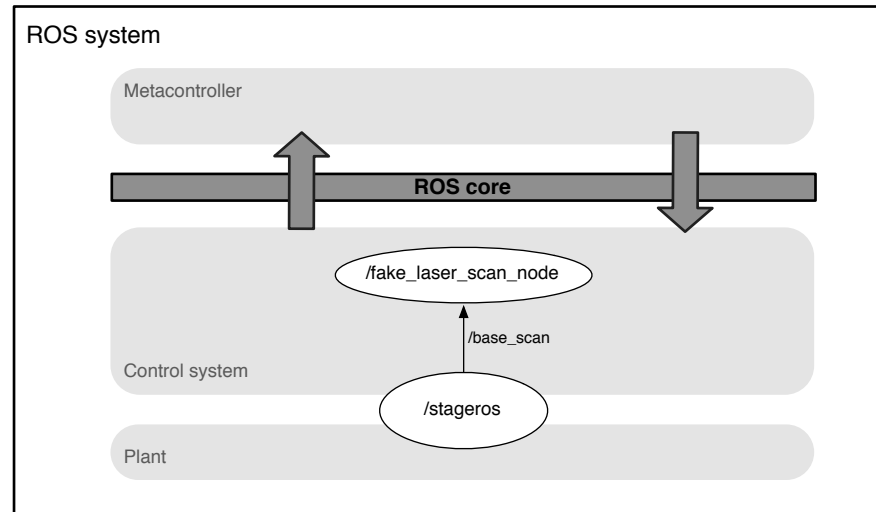


Figure B.4: The testbed 2b. The `/stageros` node contains the robotic simulation, that is the controlled system.

Testbed 2c: includes the complete control system for the simulated robot.

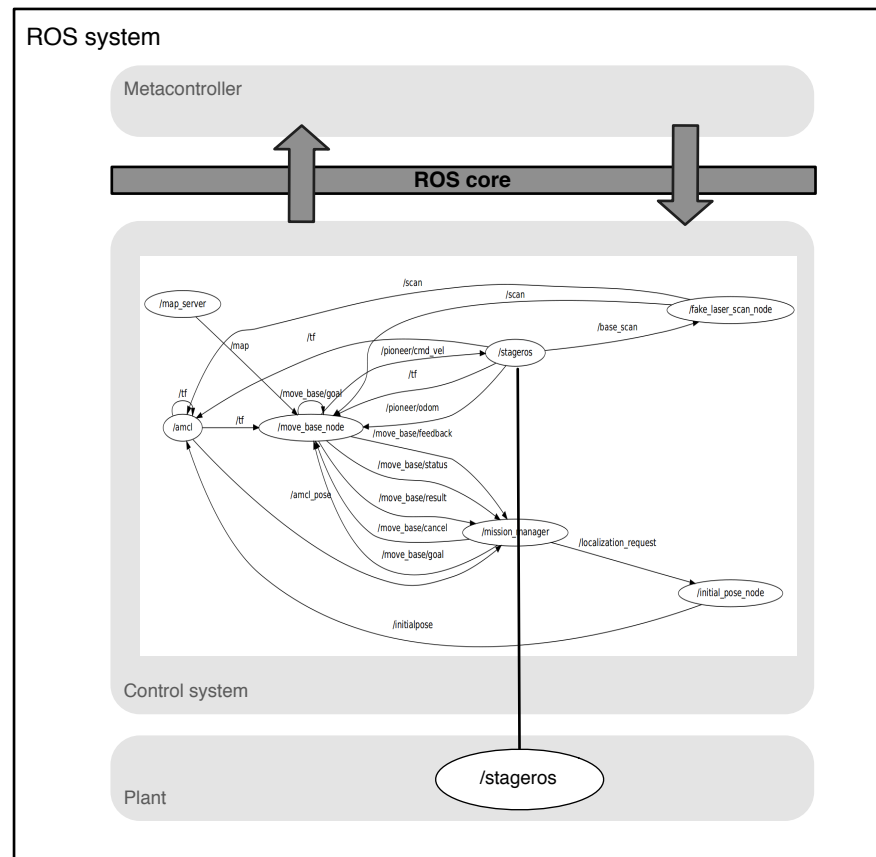


Figure B.5: The testbed 2c

Purpose of Testbed 2:

Validate OMJava with the real control system for navigation but applied to a simulated version of the Higgs robot.

Rationale: The metacontrol system in this testbed consists as in all testbeds of an instance of `OMJava.metacontrol.OMmetacontroller` (called metacontroller), but it now runs inside a rojava node to control the ROS nodes of a ROS control application in which the robot plant is a simulation. Two ROS nodes are also developed to act as the sensors and actuators of the metacontroller node, by using the ROS introspection API. `OMRosAPI` act as a wrapper of this I/O to offer the `PlantAPI` services to the metacontroller.

Expected result: OMJava library and its adapter to ROS (consisting of the projects `OMROSMetacontrol` and `OMROSDrivers_py`) are fully functioning for the simulation of Higgs using stage and ROS.

Developed artifacts: Following is a list of the new artifacts or improvements of those existing that have been produced for testbed 2.

- New RSA model in `OM.Models` with specifics for this application (ToDo)
- Improvements in `OMArchitecturalFramework` model (To update from code)
- Code:
 - Update *OMJava Eclipse project*
 - New Eclipse project `OMRosMetacontrol` : instantiates the OM1 metacontroller within a rojava node (Java): `OMMetacontroller.java` (To improve for scenario c)
 - New Eclipse project and ROS package: `OMRosDrivers_py` , containing Python nodes for actuation and sensing with the `OMMetacontroller` (Done)
 - New java packages for generating the knowledge database and initial state of the metacontroller for Testbed 2 scenarios, in the `OM.Testbed2` Eclipse project
 - New package `org.aslab.om.testbeds.knowledge` that contains knowledge about ROS components, specifically those present in the testbeds.

Due date: July 31 — Status: done scenarios a) and b) scenario c) under development

B.2.1 Testbed 2a

The control system consists of a single node: `/fake_laser_scan_node` , which is a python node that simulates the robot's laser driver.

Validation Scenario

Scenario 1	Recover laser component connected to robot
Purpose	Validate that the Components Loop is capable of re-launching a component which is not compliant with the specification required because it has a wrong configuration either in a parameter.
Description	With the system running, the baudrate parameter of the <code>/fake_laser_scan_node</code> ROS node gets a wrong value. The ComponentSECL detects this wrong state and commands the component action of reconfiguring that component. The OMRosAPI receives this action and commands the <code>/meta_actuator</code> node to KILL <code>/fake_laser_scan_node</code> and LAUNCH it with the correct configuration
Preconditions	<ul style="list-style-type: none"> Control system initially consisting of the simulated laser driver. The metacontrol system is running: <div data-bbox="709 722 1312 911" data-label="Diagram"> <pre> graph TD MC([/meta_controller]) MSN([/meta_sensor_node]) MAN([/meta_actuator_node]) MC -- "/meta_singularity" --> MSN MSN -- "/meta_referents" --> MC MC -- "/meta_action" --> MAN MAN -- "/meta_action_result" --> MC </pre> </div> Metacontrol goal: control system configuration as it is defined by the first precondition, with the parameter <code>/fake_laser_scan_node/baudrate=5600</code>
Triggering event	Laser baudrate change (e.g. the user changes the value of the baudrate parameter manually).

Basic Flow	<ol style="list-style-type: none"> 1. In ComponentsECL.perceive() the new configuration of the <code>/fake_laser_scan_node</code> is updated in the <code>estimated_state</code>, so the perceived baudrate is 5600 2. In ComponentsECL.evaluate() the mismatch between the estimated state of <code>/fake_laser_scan_node</code> and the desired specification of it in the goal produces an entry in the error signal, consisting of the difference in between the baudrate for the laser in the goal and the perceived one. 3. In ComponentsECL.control() the action to RECONFIGURE the <code>/fake_laser_scan_node</code> component ($A1[RECONFIGURE, < _fake_laser_scan_node, baudrate = 5600 >]$) is produced and sent to OMRosAPI 4. In OMRosAPI.manager implements the action A1 as two commands to the OMRosDrivers.py/meta_actuator_node, one to kill the current <code>/fake_laser_scan_node</code> ($C1[KILL, < _fake_laser_scan_node >]$), and another one to launch a new instance of the laser node with baudrate=5600 which is to be executed after completion of the first one ($C2[LAUNCH, < _fake_laser_scan_node.py >, depends = C1]$). 5. The action commands are executed sequentially: <ol style="list-style-type: none"> a. The command C1 is executed b. The feedback about success of C1 is received (in OMRosAPI.manager) c. The command C2 is executed d. The feedback about success of C2 command is received (in OMRosAPI.manager) 6. The feedback about the action A1 ($FB1[A1, SUCCESS]$) is produced in OMRosAPI.manager and added to the buffer 7. In ComponentsECL.perceive() the action feedback FB1 is processed <i>and matched against the perception of components</i>
Postconditions	Same control system that in the preconditions
Scenario execution instructions	<ol style="list-style-type: none"> 1. Execute in a terminal: \$ roslaunch OM_Testbed2 testbed2a.launch 2. from within Eclipse, run the RosRun configuration for project OMRosMetacontrol 3. in another terminal, execute rosparam set <code>/fake_laser_scan_node/baudrate 19200</code>


B.2.2 Testbed 2b


The control system consists of two nodes: `/stageros`, which simulates the robot, and `/fake_laser_scan_node`, which is a python node that simu-


lates the robot's laser driver.

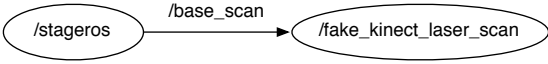
Validation Scenarios

Following we describe the scenarios corresponding to the validation tests we have realised to demonstrate the functionality rendered by the metacontrol in this testbed:

Scenario 1	Recover laser component connected to robot
Purpose	Validate that the Components Loop is capable of re-launching a component which is not compliant with the specification required because it has a wrong configuration either in a parameter or a port (wrong connection).
Description	With the system running, the baudrate parameter of the <code>/fake_laser_scan_node</code> ROS node gets a wrong value. The ComponentSECL detects this wrong state and commands the component action of reconfiguring that component. The OMRosAPI receives this action and commands the <code>/meta_actuator</code> node to KILL <code>/fake_laser_scan_node</code> and LAUNCH it with the correct configuration
Preconditions	<ul style="list-style-type: none"> Control system initially consisting of 2 ROS nodes, the stage simulation and the simulated laser driver.  <pre> graph LR A([/stageros]) -- /base_scan --> B([/fake_laser_scan_node]) </pre> <ul style="list-style-type: none"> The Metacontrol system is up and running. Metacontrol goal: control system configuration as it is defined by the first precondition, with the parameter <code>/fake_laser_scan_node/baudrate = 5600</code>
Triggering event	Laser baudrate change (e.g. the user changes the value of the baudrate parameter manually).

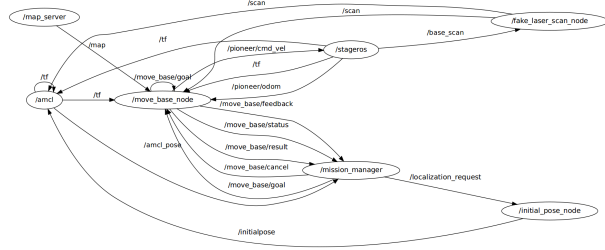
Basic Flow	<ol style="list-style-type: none"> 1. In ComponentsECL.perceive() the new configuration of the <code>/fake_laser_scan_node</code> is updated in the <code>estimated_state</code>, so the perceived baudrate is 5600 2. In ComponentsECL.evaluate() the mismatch between the estimated state of <code>/fake_laser_scan_node</code> and the desired specification of it in the goal produces an entry in the error signal, consisting of the difference in between the baudrate for the laser in the goal and the perceived one. 3. In ComponentsECL.control() the action to RECONFIGURE the <code>/fake_laser_scan_node</code> component ($A1[RECONFIGURE, < /fake_laser_scan_node, baudrate = 5600 >]$) is produced and sent to OMRosAPI 4. In OMRosAPI.manager implements the action A1 as two commands to the OMRosDrivers.py/meta_actuator_node, one to kill the current <code>/fake_laser_scan_node</code> ($C1[KILL, < /fake_laser_scan_node >]$), and another one to launch a new instance of the laser node with baudrate=5600 which is to be executed after completion of the first one ($C2[LAUNCH, < fake_laser_scan_node.py >, depends = C1]$). 5. The action commands are executed sequentially: <ol style="list-style-type: none"> a. The command C1 is executed b. The feedback about success of C1 is received (in OMRosAPI.manager) c. The command C2 is executed d. The feedback about success of C2 command is received as a <code>/meta_action_result MAR1[C2, SUCCESS]</code> (in OMRosAPI.manager) 6. The feedback about the action A1 ($FB1[A1, SUCCESS]$) is produced in OMRosAPI.manager and added to the buffer 7. In ComponentsECL.perceive() the action feedback FB1 is processed <i>and matched against the perception of components</i>
Postconditions	<p>Same control system that in the preconditions</p>  <pre> graph LR A(/stageros) -- /base_scan --> B(/fake_laser_scan_node) </pre>
Scenario execution instructions	<ol style="list-style-type: none"> 1. Execute in a terminal: \$ roslaunch OM_Testbed2 testbed2b.launch 2. from within Eclipse, run the RosRun configuration for project OMRosMetacontrol 3. in another terminal, execute rosparam set <code>/fake_laser_scan_node/baudrate 19200</code>

Scenario 2	Reconfigure to change laser for kinect because of laser permanent failure
Purpose	Validate that the metacontrol is capable of setting a new configuration because a component is no longer available.
Description	With the system running, the laser is "disconnected" (to simulate this the <code>/fake_laser_node</code> is killed and the file simulating the laser port is eliminated). Because of the second action, the metacontrol cannot successfully re-launch the <code>/fake_laser_scan_node</code> , so an alternative configuration is sought and found in the knowledge at the functional level, where there is an alternative FDesign for the function of providing laser lectures, which consists of connecting the <code>/fake_kinect_laser_scan</code> to the robot (simulated by <code>/stageros</code>)
Preconditions	<ul style="list-style-type: none"> Control system  <pre> graph LR A([/stageros]) -- /base_scan --> B([/fake_laser_scan_node]) </pre> <ul style="list-style-type: none"> Metacontrol System up and running. Metacontrol Goal: O1 - obtain laser scans continuously.
Triggering event	Kill laser node to unrecoverable state (e.g. manually by human intervention).

Basic Flow	<p>The basic flow starts with events 1 to 5.c of Scenario 1, then it diverges:</p> <ol style="list-style-type: none"> 5. d. The feedback message MA1 (from <code>/meta_actuator_node</code>) about the ROS command C2 to LAUNCH the <code>/fake_laser_scan_node</code> contains a FAILURE result. 6. MA1 is processed by the corresponding ROSTaskExecutor and the ReconfigurationManager to generate an entry in OMRosAPI.feedbackBuffer with the failure of the RECONFIGURATION of the component ($FB1[A1, FAILURE]$). 7. In ComponentsECL.proprioception() the feedback FB1 is processed producing the following perceptions: <ol style="list-style-type: none"> a. The component subgoal (specification atom) of having the <code>/fake_laser_scan_node</code> becomes in ERROR ($CG1[ERROR]$). b. The ComponentClass for the <code>/fake_laser</code> becomes UNAVAILABLE. 8. At FunctionaleCL.perceive() $CG1[ERROR]$ causes the FunctionGrounding that uses the <code>/fake_laser_scan_node</code>, and subsequently the root objective of having laser reading to be in error. 9. (FunctionaleCL.evaluate() doesn't change anything since there are no branches in the functional hierarchy tree, having just one objective) 10. In FunctionaleCL.control() an alternative FunctionDesign for O1 is sought and found, by considering that it cannot make use of the <code>/fake_laser_scan.py</code> component class (because $CG1[ERROR]$), and the FD2 is instantiated and sent as a FunctionalAction FA1 to the ComponentsECL 11. The ComponentsECL converts FA1 in its new ComponentGoal CG2 12. a normal cycle of operation at the ComponentECL produces the final configuration of the control system shown in the postconditions. <i>complete</i>
Postconditions	 <pre> graph LR A(/stageros) -- /base_scan --> B(/fake_kinect_laser_scan) </pre>
Scenario execution instructions	<ol style="list-style-type: none"> 1. Execute in a terminal: \$ <code>roslaunch OM_Testbed2 testbed2b.launch</code> 2. from within Eclipse, run the RosRun configuration for project OMRosMetacontrol 3. eliminate the laser_port file <i>need to find a best way to simulate this</i>

B.2.3 Testbed 2c

The control system consists of the complete navigation system, connected to a simulation of Higgs in Stage.

Scenario 2	Reconfigure to change laser for kinect because of laser permanent failure
Purpose	Validate that the metacontrol is capable of setting a new configuration because a component is no longer available.
Description	With the system running, the laser is "disconnected" (to simulate this the <code>/fake_laser_node</code> is killed and the file simulating the laser port is eliminated). Because of the second action, the metacontrol cannot successfully re-launch the <code>/fake_laser_scan_node</code> , so an alternative configuration is sought and found in the knowledge at the functional level, where there is an alternative FDesign for the function of providing laser lectures, which consists of connecting the <code>/fake_kinect_laser_scan</code> to the robot (simulated by <code>/stageros</code>)
Preconditions	<ul style="list-style-type: none"> Control system:  Metacontrol system up and running. Metacontrol Goal:
Triggering event	Kill laser node to unrecoverable state (e.g. manually by human intervention).

<p>Basic Flow <i>rehacer, es una copia del anterior</i></p>	<p>The basic flow starts with events 1 to 5.c of Scenario 1, then it diverges:</p> <ol style="list-style-type: none"> 5. d. The feedback message MA1 (from <code>/meta_actuator_node</code>) about the ROS command C2 to LAUNCH the <code>/fake_laser_scan_node</code> contains a FAILURE result. 6. MA1 is processed by the corresponding ROSTaskExecutor and the ReconfigurationManager to generate an entry in OMRosAPI.feedbackBuffer with the failure of the RECONFIGURATION of the component ($FB1[A1, FAILURE]$). 7. In ComponentsECL.proprioception() the feedback FB1 is processed producing the following perceptions: <ol style="list-style-type: none"> a. The component subgoal (specification atom) of having the <code>/fake_laser_scan_node</code> becomes in ERROR ($CG1[ERROR]$). b. The ComponentClass for the <code>/fake_laser</code> becomes UNAVAILABLE. 8. At FunctionaleECL.perceive() $CG1[ERROR]$ causes the FunctionGrounding that uses the <code>/fake_laser_scan_node</code>, and subsequently the root objective of having laser reading to be in error. 9. (FunctionaleECL.evaluate() doesn't change anything since there are no branches in the functional hierarchy tree, having just one objective) 10. In FunctionaleECL.control() an alternative FunctionDesign for O1 is sought and found, by considering that it cannot make use of the <code>/fake_laser_scan.py</code> component class (because $CG1[ERROR]$), and the FD2 is instantiated and sent as a FunctionalAction FA1 to the ComponentsECL 11. The ComponentsECL converts FA1 in its new ComponentGoal CG2 12. a normal cycle of operation at the ComponentECL produces the final configuration of the control system shown in the postconditions. <i>complete</i>
<p>Postconditions</p> <p>Scenario execution instructions</p>	<ol style="list-style-type: none"> 1. Execute in a terminal: \$ <code>roslaunch OM_Testbed2 testbed2c.launch</code> 2. from within Eclipse, run the RosRun configuration for project OMRosMetacontrol 3. eliminate the laser_port file <i>need to find a best way to simulate this</i>

B.3 Testbed 3: navigation of the real mobile robot

B.3.1 Scenario 1

See activity diagram in figure B.6.

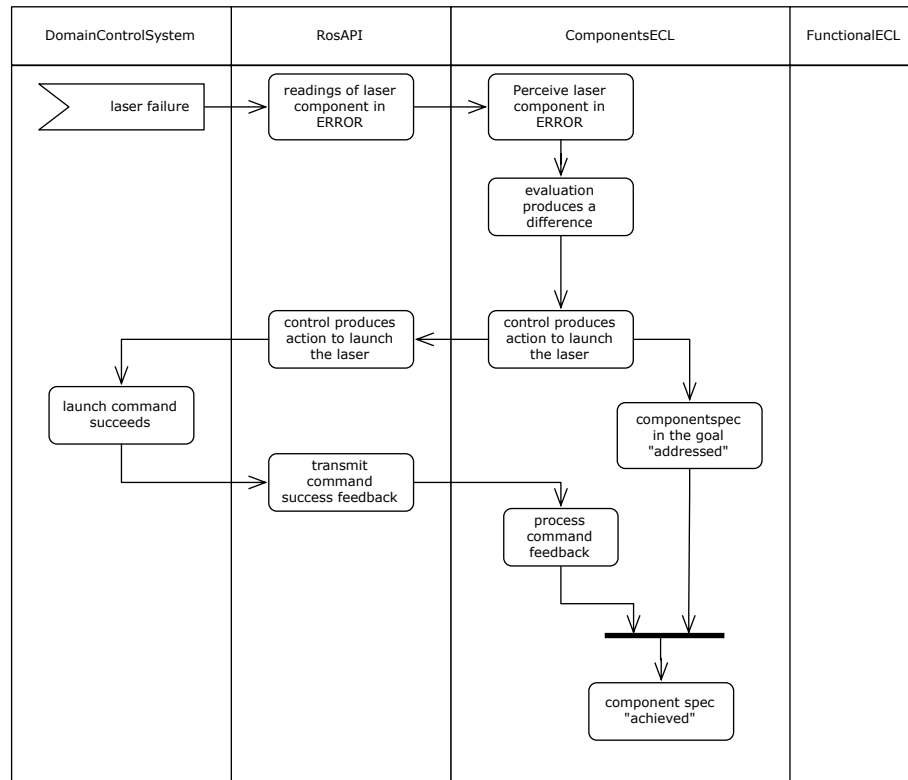


Figure B.6: Scenario in which the laser fails and it is re-started by the ComponentsECL. Boxes are events, rather than activities arrows show casual relationships between events

B.3.2 Scenario 2

See activity diagram in figure B.7

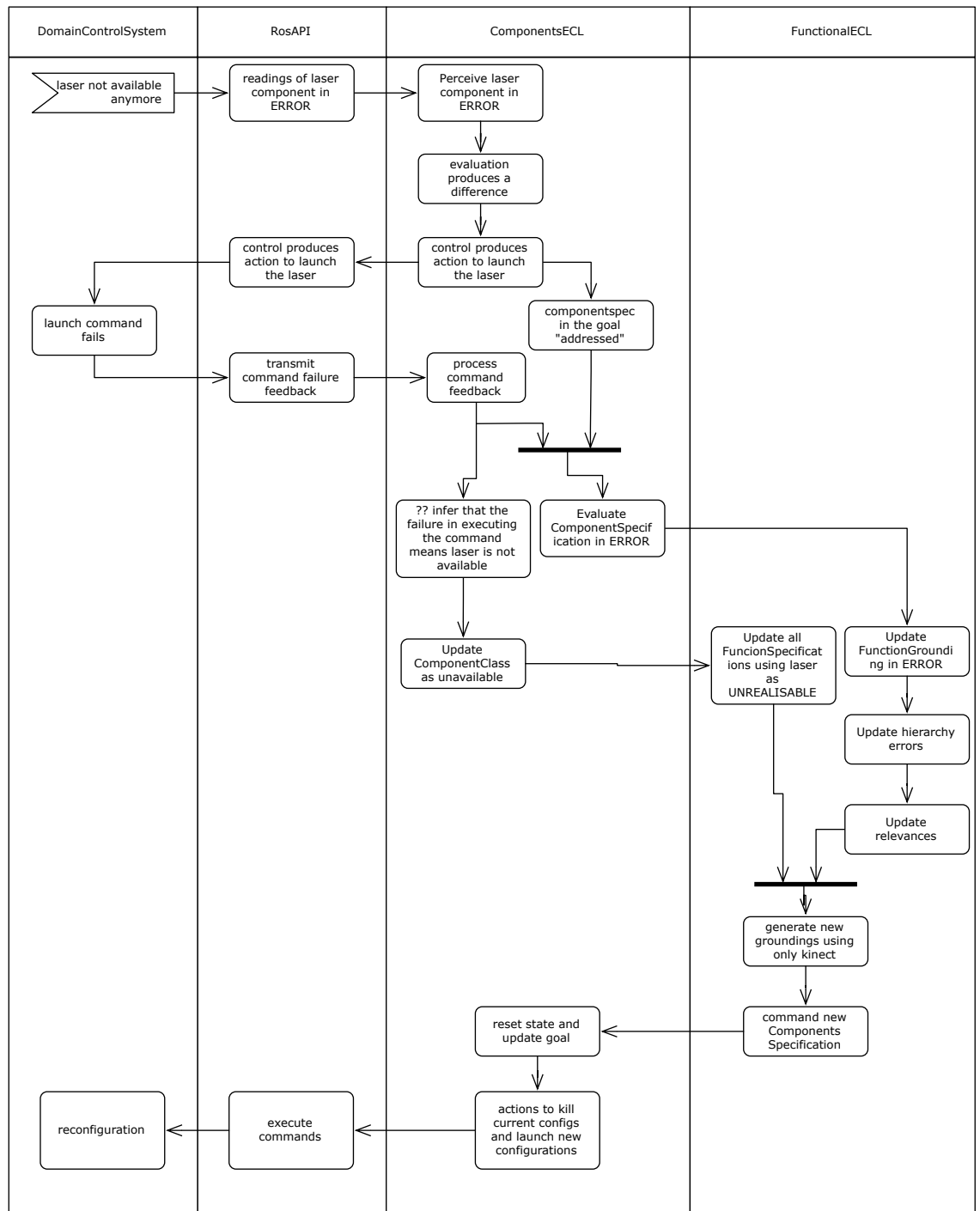


Figure B.7: Scenario in which the laser fails and it gives an error when trying to launch it again. Boxes are events, rather than activities arrows show casual relationships between events

Bibliography

- [Garage, 2011] Garage, W. (2011). Robot operating system.
- [Gerkey et al., 2003] Gerkey, B. P., Vaughan, R. T., and Howard, A. (2003). The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, pages 317–323, Coimbra, Portugal.
- [Hernandez, 2013] Hernandez, C. (2013). *CH's thesis*. PhD thesis.
- [Marder-Eppstein et al., 2010] Marder-Eppstein, E., Berger, E., Foote, T., Gerkey, B., and Konolige, K. (2010). The office marathon: Robust navigation in an indoor office environment. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 300–307.
- [Quigley et al., 2009] Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*.

Title: OMJava

Subtitle: Java library for the OM Architectural Framework

Author: Carlos Hernández

Date: March 16, 2013

Reference: R-2012-008 v 0.4 Draft

URL: <http://www.aslab.org/documents/controlled/ASLAB-A-2012-008.pdf>

© 2012 ASLab

Autonomous Systems Laboratory

UNIVERSIDAD POLITÉCNICA DE MADRID
C/JOSÉ GUTIÉRREZ ABASCAL, 2
MADRID 28006 (SPAIN)

