UNIVERSITY OF STUDY OF BERGAMO

DEPARTMENT OF ENGINEERING

Computer Engineering

Self-Adaptive Onboard Managing System

Yamuna Maccarana - matr.1014766

Umberto Paramento - matr. 1015316

ACADEMIC YEAR 2014/2015

Contents

List of Figures

Chapter 1

Introduction

This project has been developed as completion to the course of Design and Algorithms with Professor Scandurra, at the University of Bergamo, placed in Dalmine. It investigates interesting aspects of self-adaptation models and architectures.

Through the building of this project, important matters of a greater planning (such as software distribution for functional components and data exchange, requirements management and use case modelling, development of software components, design patterns "at the level of software architecture", unit testing and coverage of the components of a software application tool for the calculation of metrics and for the re factoring of the software) and of a smaller planning (such as detailed analysis of each component and definition of algorithms and data structures, code development cycle of the algorithm, implementation of algorithms, inheritance and abstract data types in Java, fundamental algorithms and data structures - lists, stacks, queues -, methodologies for designing algorithms - especially dynamic programming - and exception management) have been examined in depth.

As far as it is a completion to the course, this project has been planned as a robust and flexible starting point for further extensions and adaptation. Through the use of the The Task Component Architecture [2][3], this project as been conceived

as an easily implementable framework to provide the user a great choice of scopes of application. Thanks to the definition of its properties, each component is quickly configurable to run autonomously.

With this manual the user will be able to understand each step of the project and, moreover, they will be capable to extend or modify this tool for a specific need.

Figure 1.1: Universitatis bergomensis studium logo.

1.1 Acknowledgements

First of all, we would like to thank our teacher, Patrizia Scandurra, Associate Professor at University of Bergamo, who started us toward this project, stimulated us and attended upon us for the whole duration of our work. Then, we mutually want

to thank ourselves, because we found a friend more than a workmate in each other, and this made things much easier and sometimes even hilarious.

Chapter 2

Foreword

2.1 Aims

The aim of this project is to provide a robust and flexible system for self-adaptive periodic components managed through MAPE-K classes. In chapter 3 background knowledges are exhaustively explained.

This project has been conceived for a robotic environment. Through the use of the tool provided by this project, new robust and flexible self-adaptive applications can be created. Also, this tool finds applications in data management applications, such as relational database management. In the end, this project can be used for web access, thanks to already existing SCA functionalities. This makes the Self-Adaptive Onboard Managing System (SAOMS) the missing link between the top and the bottom of a complete software application.

2.2 Equipment

Attached to this documentation there some software containing:

- lib: unibg.saoms.jar
- src: source code of unibg.saoms

- JavaCode: testing and case study

In case of loss, it is possible to find repositories and documentation online [1].

Chapter 3

Background

Before getting to the heart of the matter, it is helpful to introduce basic concepts that are essential to be known and understood. Here are overviews of the Service Component Architecture and the Task Component Architecture.

3.1 Service Component Architecture

Service Component Architecture (SCA) [5] is a set of specifications intended for the development of applications using a Service Oriented Architecture (SOA) [7], which defines how computing entities interact to perform work for each other. Moreover, SCA is an XML-based metadata model that describes the relationships and the deployment of services independently from SOA platforms and middleware Application Programming Interfaces (APIs). The specification of a SCA component is defined as an XML-based model in terms of provided services, required services (called references), and properties. For this project's purposes, the specification of SCA components is defined by Java interfaces and their implementation is provided by Java classes.

3.2 Extensible Markup Language

Extensible Markup Language (XML) [4] is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. A markup language is a modern system for annotating a document in a way that is syntactically distinguishable from the text. Through these languages it is possible to define a set of specification to describe a text's representation mechanisms (structural, semantic, etc.). The great advantage is that, using standard conventions, each file is reusable on different work stations and it can be implemented using different languages.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
3      name="robotEnvironment"
4      targetNamespace="http://robotEnvironment">
5
6    <component name="RobotComponent">
7      <implementation.java class="resources.robotEnvironment.Robot"/>
8      <property name="ID">TestRobot001</property>
9      <property name="nickname">Discovery</property>
10     <property name="weight">20</property>
11     <property name="height">50</property>
12     <property name="price">499.99</property>
13     <service name="move">
14       <interface.java interface="resources.robotEnvironment.RobotInterface"/>
15     </service>
16     <reference name="updatePosition">
17         <interface.java interface="resources.robotEnvironment.GPSInterface"/>
18     </reference>
19   </component>
20
21 </composite>
```

Figure 3.1: XML file defining a robot composite

A graphic view of the case study can be found in fig.3.2. In this figure the robot component is defined as shown in listing 3.1: it is possible to see the type of component (RobotComponent), its implementation (Java implementation - see the small "J" symbol in the top-left of the component), its properties (ID, nickName, weight, height and price) and its service and reference. The composite is named as

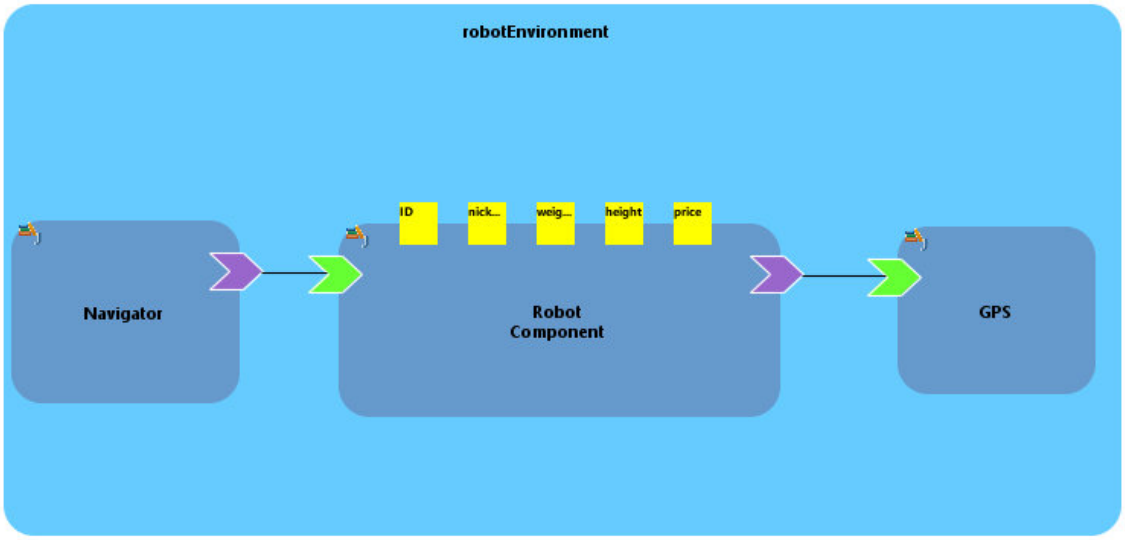

Figure 3.2: Graphic view of a robot composite with GPS and Navigator components

"robotEnvironment". Two other components are listed in the XML file as Navigator and GPS, with their reference (Navigator) and service (GPS), their implementation (Java) and no properties. Next, it will be explained how to use Java to implement the XML files.

3.3 Java Platform

Java [8] is a computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code which runs on one platform does not need to be re-compiled to run on another.

3.3.1 Java Concurrent

The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the

Java class libraries. In particular java.util.concurrent is a package that contains utility classes commonly useful in concurrent programming. It includes a few small standardized extensible frameworks, as well as some classes that provide useful easily implementable functionalities.

Component Used

Executor An Executor is a simple standardized interface for defining custom thread-like subsystems, including thread pools, asynchronous IO, and lightweight task frameworks. The ScheduledExecutorService subinterface and associated interfaces add support for delayed and periodic task execution.

Future A Future represents the result of an asynchronous computation. Its methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using getter method when the computation has completed, blocking if necessary until it is ready. The ScheduledFuture subinterface is the result of scheduling a task with a ScheduledExecutorService, it is a delayed result-bearing action that can be cancelled.

TimeUnit A TimeUnit represents time durations at a given unit of granularity and provides utility methods to convert across units, and to perform timing and delay operations in these units. It does not maintain time information, but only helps organize and use time representations that may be maintained separately across various contexts. A TimeUnit is mainly used to inform time-based methods how a given timing parameter should be interpreted.

3.3.2 Java Messaging Service

Java Message Service (JMS) is an application program interface (API) that provides a common way for Java programs to create, send, receive and read an enterprise messaging system messages. The JMS component allows messages to be sent to (or consumed from) a JMS Queue or Topic. A JMS application is composed of the following parts:

- JMS Provider - a messaging system that implements the JMS API in addition to the other administrative and control functionality required of a full-featured messaging product;
- JMS Clients - the Java language programs that send and receive messages;
- Messages - objects that are used to communicate information between the clients of an application;
- Administered Objects - provider-specific objects that clients look up and use to interact with a JMS provider;
- Non-JMS Clients - clients that use a message system's native client API instead of the JMS API. If the application predated the availability of the JMS API, it is likely that it will include both JMS clients and non-JMS clients.

Component Used

Connection A Connection object is a client's active connection to its JMS provider that allocates provider resources outside the Java Virtual Machine (JVM). It encapsulates an open connection with a JMS provider. Connections support

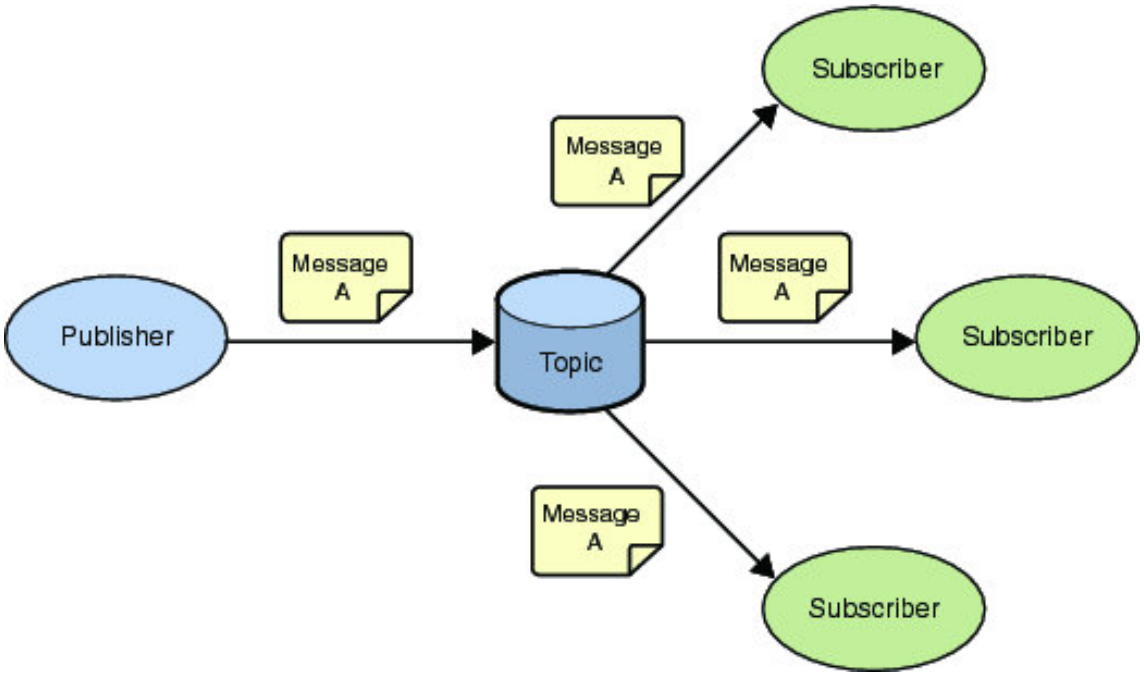

Figure 3.3: Publish/subscribe pattern

concurrent use. Moreover, it typically represents an open TCP/IP socket between a client and the service provider software.

- Its creation is where client authentication takes place;
- It can specify a unique client identifier;
- It provides a ConnectionMetaData object;
- It supports an optional ExceptionListener object.

ConnectionFactory A ConnectionFactory object is a JMS administered object and supports concurrent use. JMS administered objects are objects containing configuration information that are created by an administrator and later used by JMS clients. Indeed, this object encapsulates a set of connection configuration parameters that has been defined by an administrator and that are later used by clients to create a connection with a JMS provider. This strategy provides several benefits:

- It hides provider-specific details from JMS clients;
- It abstracts administrative information into objects in the Java programming language ("Java objects") that are easily organized and administered from a common management console;
- Since there will be JNDI (Java Naming and Directory Interface) providers for all popular naming services, JMS providers can deliver one implementation of administered objects that will run everywhere. In this case it is not necessary to use a name resolution strategy since the working context is restricted and there is a direct addressing through IP.

MessageConsumer MessageConsumer is the parent interface for all message consumers. A client uses a MessageConsumer object to receive messages from a destination. A MessageConsumer object is created by passing a Destination object to a messageconsumer creation method supplied by a session. A client may either synchronously receive a message consumer's messages or have the consumer asynchronously deliver them as they arrive. For synchronous receipt, a client can request the next message from a MessageConsumer using one of its receive methods. There are several variations of receive that allow a client to poll or wait for the next message. The method receiveNoWait is an asynchronous reception method, so the Task will not get stuck waiting for a message and its periodicity will be maintained.

MessageProducer A client uses a MessageProducer object to send messages to a destination. A MessageProducer object is created by passing a Destination object to a messageproducer creation method supplied by a session. A client also has the option of creating a MessageProducer without supplying a destination. In

this case, a destination must be provided with every send operation. A typical use for this kind of message producer is to send replies to requests using the request's JMSReplyTo destination. A client can specify a default delivery mode, priority, and time to live for messages sent by a MessageProducer. It can also specify the delivery mode, priority, and time to live for an individual message. A JMS provider should do its best to expire messages accurately; however, the JMS API does not define the accuracy provided.

ObjectMessage An ObjectMessage object is used to send a message that contains a serializable object in the Java programming language ("Java object"). It inherits from the Message interface and adds a body containing a single reference to an object. Only Serializable Java objects can be used. If a collection of Java objects must be sent, one of the Collection classes provided since JDK 1.2 can be used, e.g. Stack and Vector are not supported.

Session A Session object is a single-threaded context for producing and consuming messages. Although it may allocate provider resources outside the Java Virtual Machine (JVM), it is considered a lightweight JMS object. A session serves several purposes:

- It is a factory for its message producers and consumers;
- It supplies provider-optimized message factories;
- It is a factory for TemporaryTopics and TemporaryQueues;
- It provides a way to create Queue or Topic objects for those clients that need to dynamically manipulate provider-specific destination names;

- It supports a single series of transactions that combine work spanning its producers and consumers into atomic units;
- It defines a serial order for the messages it consumes and the messages it produces;
- It retains messages it consumes until they have been acknowledged;
- It serializes execution of message listeners registered with its message consumers;
- It is a factory for QueueBrowsers.

A session can create and serve multiple message producers and consumers. One typical use is to have a thread block on a synchronous MessageConsumer until a message arrives. The thread may then use one or more of the Session's MessageProducers. If a client desires to have one thread that produces messages while others consume them, the client should use a separate session for its producing thread. It should be easy for most clients to partition their work naturally into sessions. This model allows clients to start simply and incrementally add message processing complexity as their need for concurrency grows.

Topic A Topic object encapsulates a provider-specific topic name. It is the way a client specifies the identity of a topic to JMS API methods. For those methods that use a Destination as a parameter, a Topic object may be used as an argument. A Topic can be used to create a MessageConsumer and a MessageProducer by calling:

- Session.CreateConsumer (Destination destinationName);
- Session.CreateProducer (Destination destinationName).

Many publish/subscribe providers group topics into hierarchies and provide various options for subscribing to parts of the hierarchy. The JMS API places no restriction on what a Topic object represents.

All the above arguments are summarized in fig.3.3.

3.3.3 APACHE ACTIVEMQ

Apache ActiveMQ is an open source message broker which fully implements the Java Message Service 1.1. It is designed for the purpose of sending messages between two applications, or two components inside one application.

BrokerService A BrokerService manages the lifecycle of an ActiveMQ Broker. It consists of a number of transport connectors, network connectors and a bunch of properties which can be used to configure the broker as it is created.

3.4 Robot Operating System

The Robot Operating System (ROS) [9] is a component-based infrastructures provided to control lower part of robots (hardware). It provides libraries and tools to help software developers create robot applications.

As completion of this project, ROS could be useful due to its message-based publish-subscriber peer-to-peer communication.

So, if the lower part's control is managed by ROS and the higher part, consisting of web services, is managed by SCA, what is missing is an integration between the two parts (see fig.3.4): this is why the Task Component Architecture is needed. It does not modify anything about SCA and ROS, but instead it creates a new dynamic software architecture.

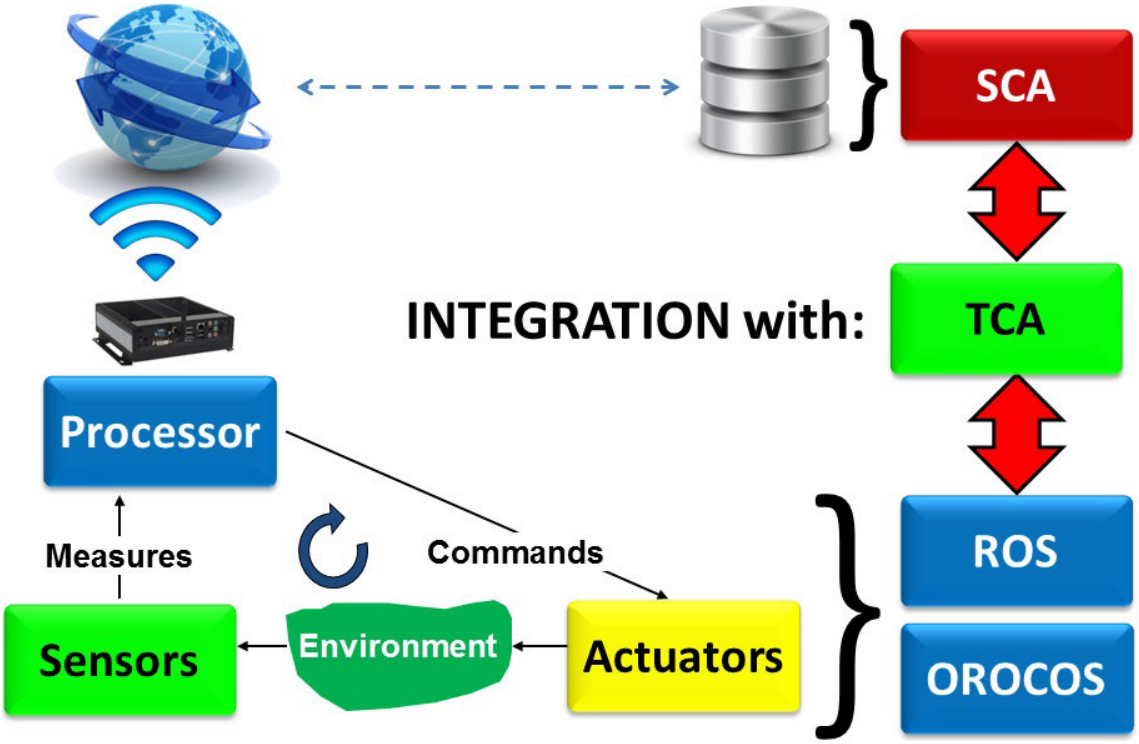

Figure 3.4: The TCA role.

3.5 Task Component Architecture

The Task Component Architecture (TCA) [2][3] is a framework implementing SCA components (see fig.3.5) that can be seamlessly integrated with robotic software control systems.

In other words, SCA uses TCA for the development of concurrent event-driven systems. It has been conceived as a Java library that executes autonomously its service without requiring client components to invoke them.

Periodic Task Management The periodic task management relies on Java concurrent: the idea is to have one dedicated autonomous process for each component.

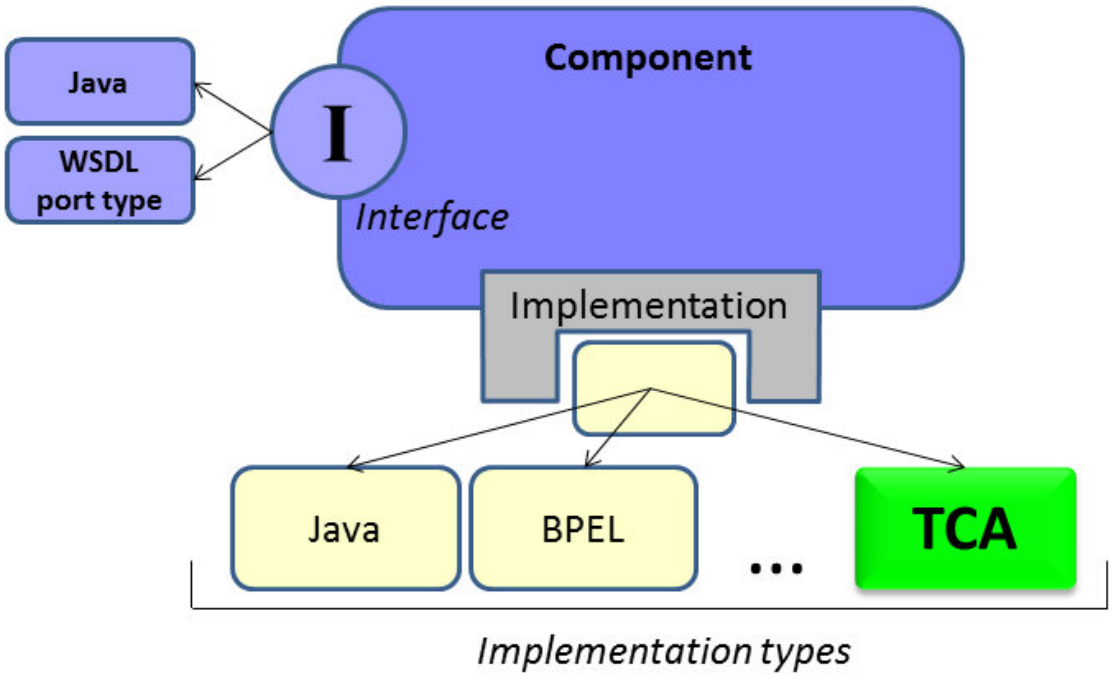

Figure 3.5: SCA implemented by TCA implementation.

Asynchronous Publish-Subscriber pattern communication The messaging system is based on a publish-subscriber pattern through the use of Java Messaging Service.

Thanks to these features, TCA consents a seamless integration with robotic software control systems. In fact, TCA is a java library using the java.util.concurrent library for the management of periodic tasks (specifically through: Future, scheduleFuture..). The class diagram can be seen in fig.3.6. The choice of relying on JMS facilitates the communication through the use of topics in a publish-subscriber pattern.

The main class of the library is the Task class that implements TaskInferface. TaskInterface presents all the methods needed to schedule and unschedule tasks, or to check if it is running or not. A task can be scheduled and unscheduled at runtime, it can run for a predetermined time or can be whenever interrupted from

the outside.

The mainstay of TCA is that programmers do not see anything about the underlying structure, nor JMS mechanisms, nor Java concurrent methods: the only thing that they have to do to create a specific application is to extend the Task class, as shown in fig.3.7. To implement a Task component it is enough to declare that the UserClass extends Task and implements TaskInterface. After that, two methods have to be overridden: initialize and execute. The initialize method only runs one time before the first execution, and it is overridden so that it makes the user component becoming a publisher on predetermined topics and a subscriber to others. The method execute contains the instructions run by the Task at each execution. After receiving all the incoming messages (or just the last one with the receiveLast method) the task component is ready to do some computation, such as elaborating images or following a path, and new messages are sent in the outbox queue to the selected topic (through the use of hashmaps and iterators).

In the end there is one last step to configure the Task component: properties. There are several required properties such as id, period and delay to define, as their names say, the name of the component, the period of each execution and a time to wait before starting its execution. The runOnStartup, mayInterruptIfRunning and useJMS are Boolean properties defining if a Task start its execution at the start of the program, if it may be interrupted while running and if it is allowed to use JMS. This last property prepare the system for the use of JMS, if it is false and the component tries to use JMS an exception is thrown. Other non-required properties such as url, user, password are used in case of applications running on different workstations, which makes the TCA a dynamic remote architecture. An example can be seen in fig.3.8.

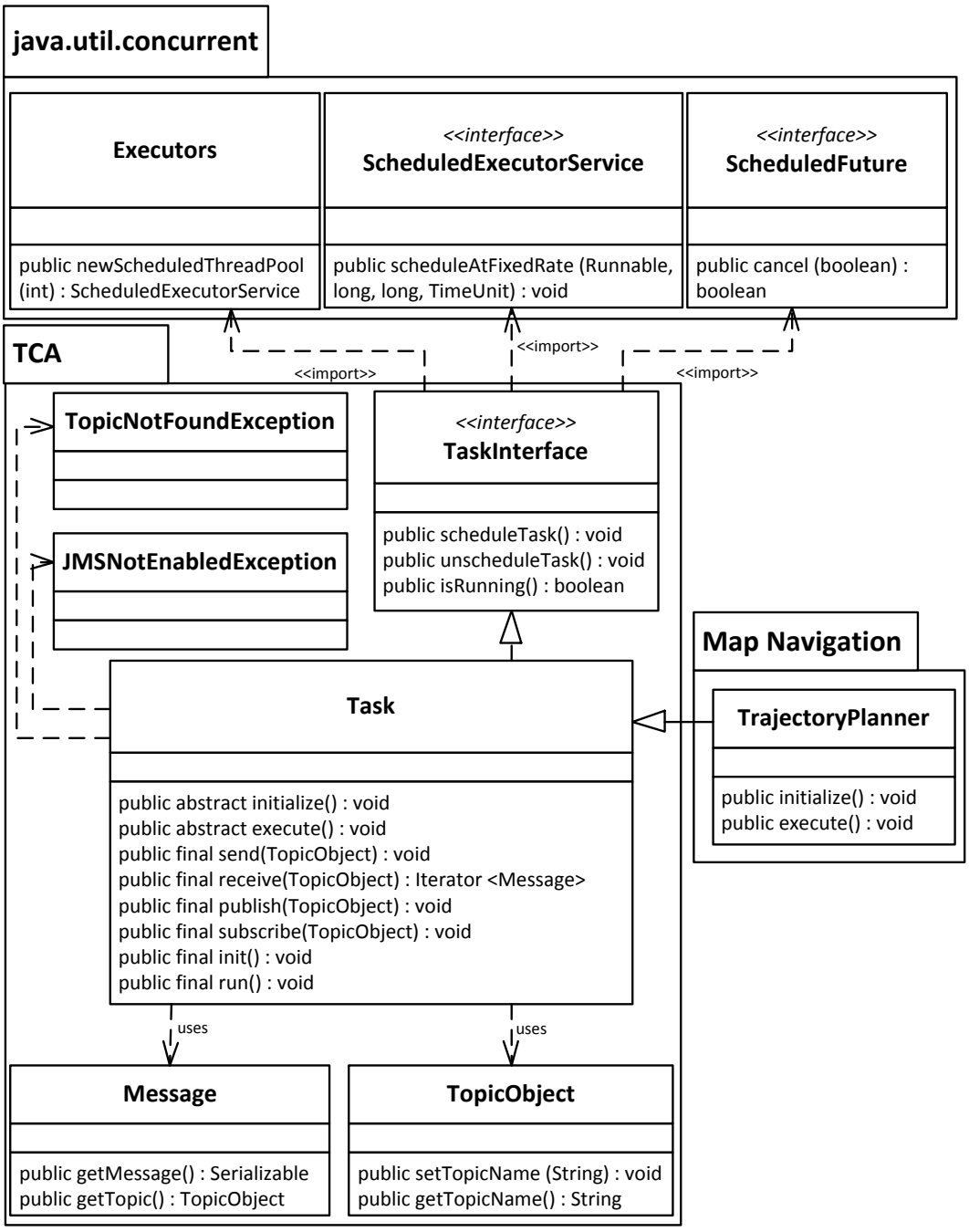

Figure 3.6: The TCA class diagram.

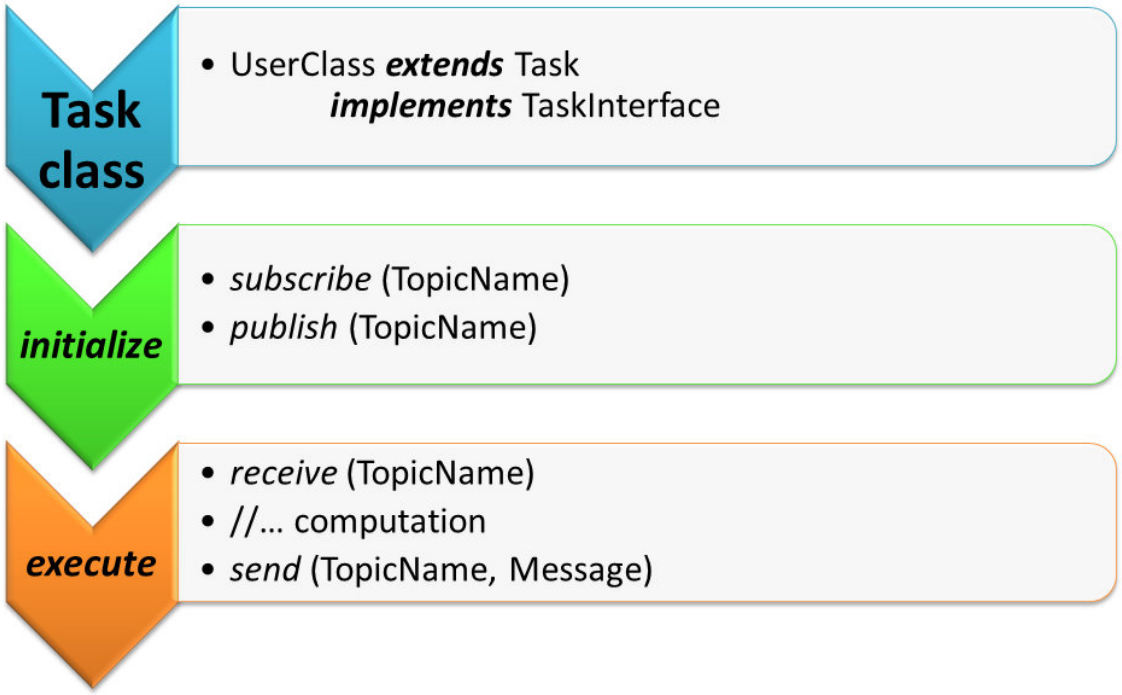

Figure 3.7: How to implement a TCA component.

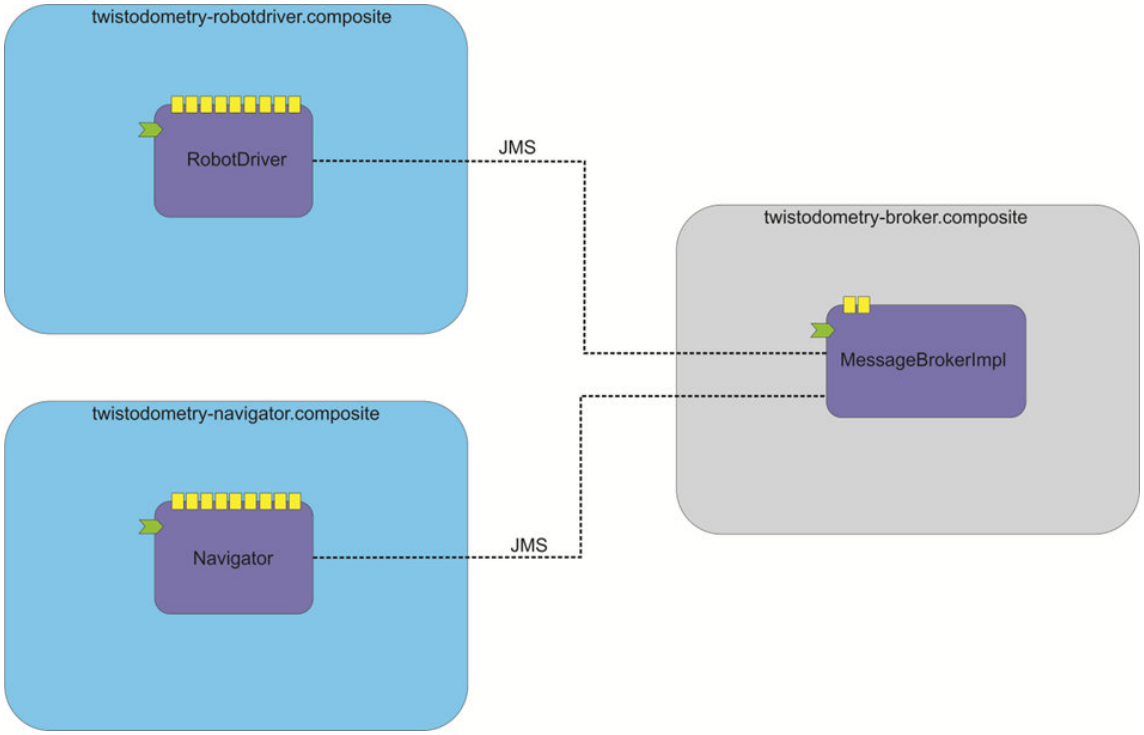

Figure 3.8: An example of TCA components running through the use of JMS.

3.6 MAPE loop

There is one last important feature to exploit, wich is integrated in this project. It is a concept of multi-agent Abstract State Machines to specify decentralized adaptation control by using MAPE (Monitor-Analyze-Plan-Execute) computations. It is a model to support techniques for validating adaptation scenarios, and getting feedback of the correctness of the adaptation logic as implemented by the managing components over the managed ones. As a proof-of-concepts, it is model and analyze a decentralized monitoring system. The following description can be seen in fig.3.9.

Managing subsystem The Managing subsystem comprises the adaptation logic and monitors the environment and the managed subsystem and adapts the latter when necessary.

Managed subsystem The Managed subsystem comprises the application logic and supports adaptation by providing probes.

M Gather particular data from underlying managed system and environment.

A Examines the collected data to determinate the system's need.

P Constructs the adaptation actions.

E Execution computation that carries out the actions.

K Mantains information of the managed system and environment.

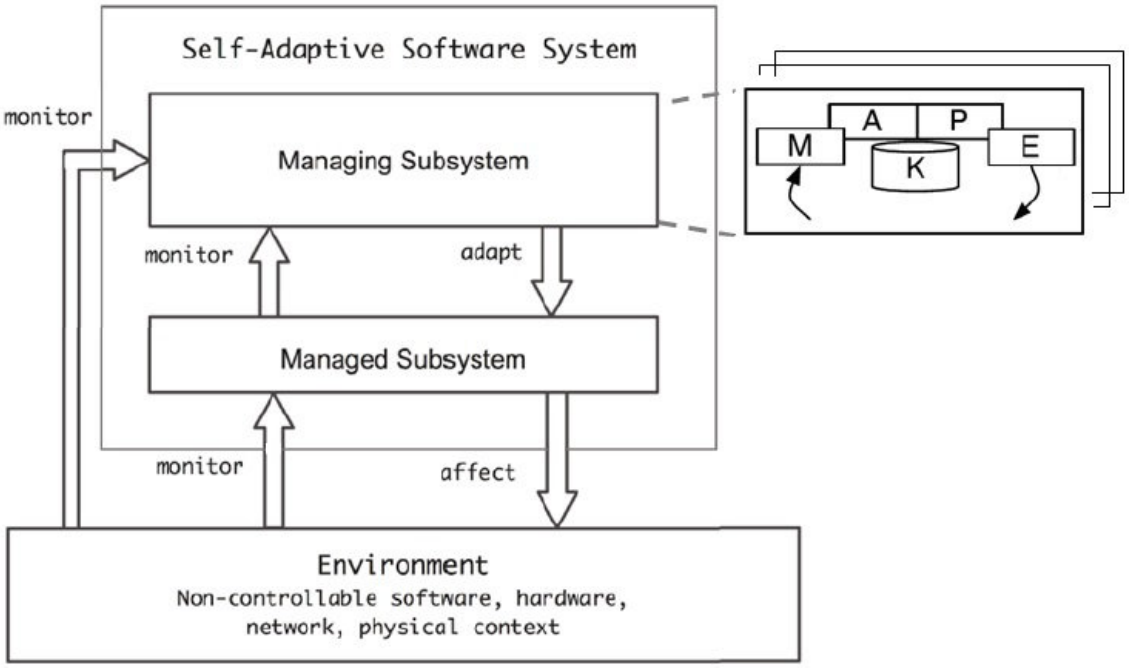

Figure 3.9: MAPE-K computation diagram.

Chapter 4

Case Study

This project deals with several locomotive robots, each one equipped with Self Adaptive OnBoard Managing Systems (SAOMS) (see chapter 6 fo further information).

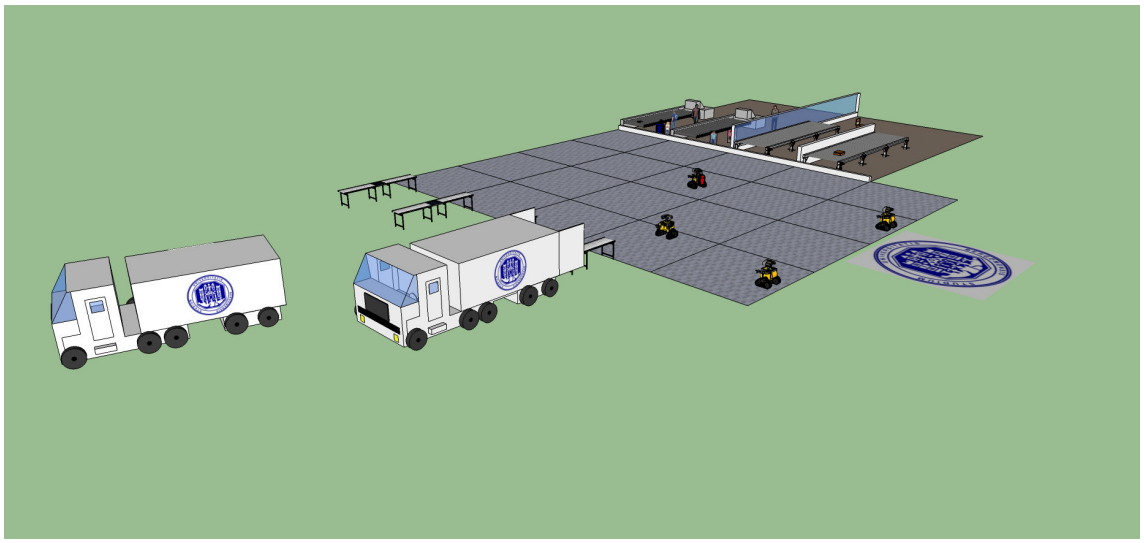

Figure 4.1: 3D rendering of the Case Study.

These robots are placed in the arrival and departures area of an airport (see fig.4.1, 4.2, 4.3). Their task is to bring luggage from one point to another. Each robot is equipped with a processing unit and a communication unit to interact with Master Managing Systems and other robots.

As robots are implemented as TCA components, each one has one dedicated

process that runs periodically. Each robot is conceived to autonomously offer repetitive services, such as being available to bring luggage from one place to another, communicating its position and relying on its dedicated Managing Systems to avoid collisions and handle critical situations.

The role or Master Managing Systems is just to manage incoming events to transform them in tasks and to define the number of robots needed to work concurrently. This last feature is the key of flexibility: depending on the workload, the system will adapt to changes and improve performances.

Flexibility comes along with robustness: in fact, one great feature of this project, is to make components able to help each other in case of need. For example, when a robot hits on a critical situation and can not move, it can call a substitute robot and gently ask it to complete its task. This means that every robot, thanks to its own processing unit, can monitor its integrity and help avoiding possible failures, and all this happens autonomously, out of the Master Managing Systems control.

In other words, flexibility comes with the ability to self-adapt to necessities (managed by Master Managing Systems), robustness comes with the ability to deal autonomously with possible robot-side failures (managed by each robot component).

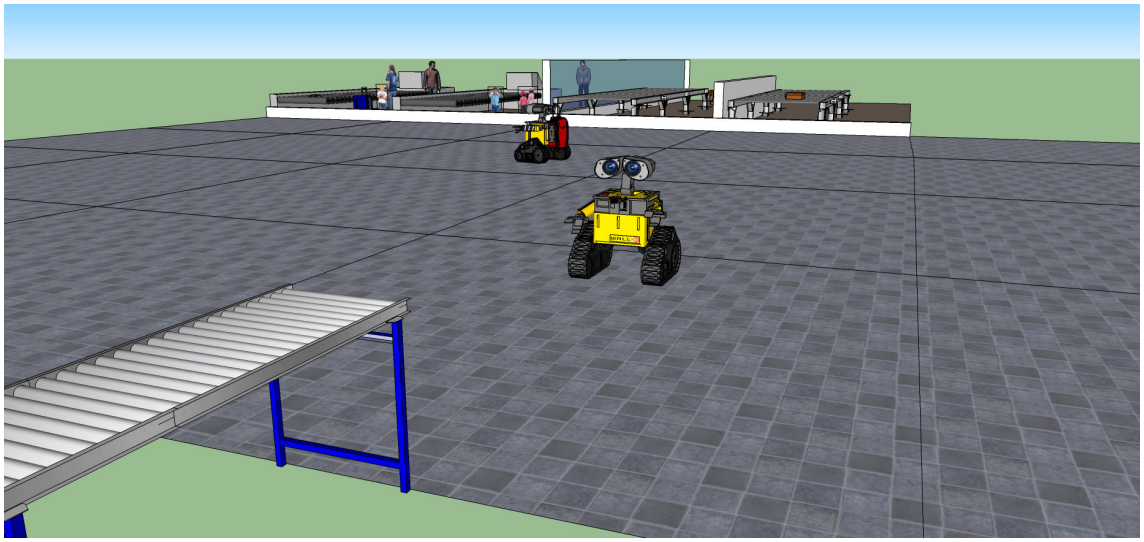

Figure 4.2: 3D rendering of the Case Study.

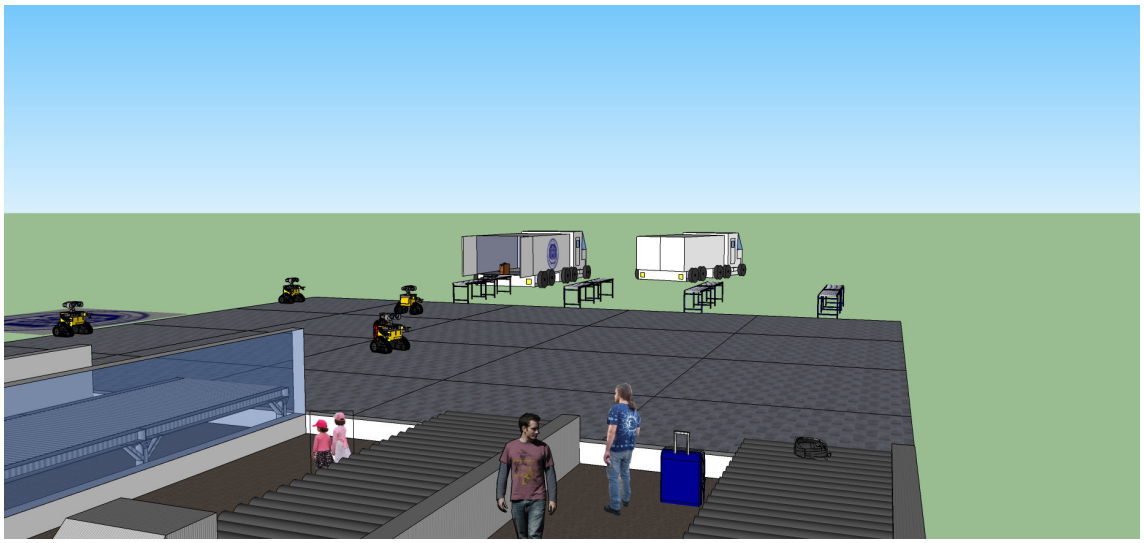

Figure 4.3: 3D rendering of the Case Study.

Chapter 5

Configuration and Folder Explanation

Software used: Eclipse Version: 4.3 Build id: M20120914-1800 Here are the steps needed to configure the working environment.

5.1 Libraries

The only libraries needed are TUSCANY [6] and TCA [3]. Please follow official instruction to install TUSCANY, and simply add the TCA .jar file in the build path of your project.

If using Eclipse, the final result of a correct configuration of the src code can be seen in fig.5.1.

5.2 Folders

Here follows the explanation of the structure of the whole project.

5.2.1 User folders

launch

In this package are contained the launchers:

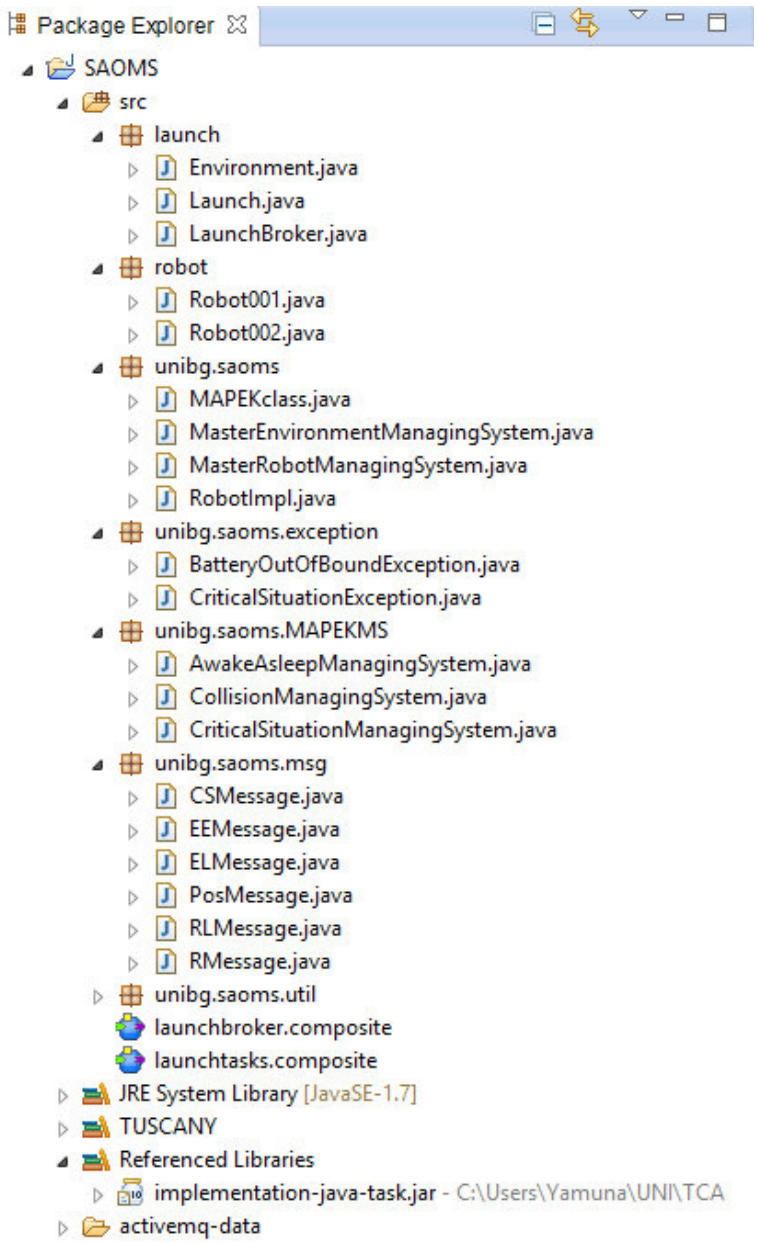

Figure 5.1: Correct configuration of the scr code.

- LaunchBroker that has to be started as the first application ever;
- Launch that is the file containing the schedulation of the tasks (MasterEnvironmentManagingSystem, MasterRobotManagingSystem, Environment and the Robots);
- Environment that is not a proper launch but contains the simulation of events such as the arrival of new pieces of luggage or airplanes.

robot

This package contains all the physical robot unities as files.

5.3 unibg.saoms jar file

unibg.saoms

This is the package containing all the MasterSystems and the father classes of Robots and MAPE-K classes:

- MAPEKclass defines the structure of each MAPE-K class, that will extend it;
- MasterEnvironmentManagingSystem is the class that monitors, analyzes and manages new environment events;
- MasterRobotManagingSystem is the virtual Master Robot which manages other Robots;
- RobotImpl is the main class of the project: it contains most of the logic and the algorithms of the entire program.

unibg.saoms.exception

In this package it is possible to find the exception to manage critical events or wrong tries to insert values by the user. Please notice that also in TCA are present also other exception that may occur during the execution.

- BatteryOutOfBoundException manages the case in which the user tries to initialize the battery with a value that is minor than 0 or major than 100;
- CriticalSituationException occurs when the Robot finds itself in a critical situation with no possibility to call other Robots for help.

unibg.saoms.MAPEKMS

All the MAPE-K Managing Systems are contained in this package:

- AwakeAsleepManagingSystem is the class managing awake/asleep requests to be sent to robots;
- CollisionManagingSystem manages robots to avoid eventual crashes;
- CriticalSituationManagingSystem is the class used to manage critical situation.

unibg.saoms.msg

This is the package in which all the messages used in this project are placed:

- CSMessage is the message containing ID and IDnumber of active robots;
- EEMessage is the message to manage new exceptional events (airplanes landing or particular workload);

- ELMessage is the message to manage new events (luggage to bring from a place to another);
- PosMessage is the message containing positions of robots;
- RLMessage is the message is used to say if a robot is active or not;
- RMessage is the message containing an indication of robots that need to asleep-/awake.

unibg.saoms.util

This is the auxiliary package containing classes that are useful to the program:

- BlockPos is the class used to divide space in blocks and identify them with x and y coordinates;
- MRcounter is used to create a list of operating robots associated with a counter (used by MasterRobotManagingSystem).

Composite Files

In the end, there are two .composite files:

- launchbroker.composite is the XML file needed for the message broker to start;
- launchtasks.composite is the XML file containing the list of every task running in the execution.

In case of simple testing, all the src code can be load as a library in a .jar file. Instead, the XML files need to be present as they are.

Chapter 6

Development

This project has been developed to provide a seamless integration between MAPE-K classes and Task Components. To do so, a Self-Adaptive Onboard System (SAOMS) has been conceived and developed.

6.1 Master/Slave policy

First of all, it is necessary to define a master/slave policy, where *Master* Managing Systems monitor external events and generate tasks and *Slaves* components perform these tasks.

There are two Master Managing Systems administrating incoming luggage to be transform in tasks for slaves and establishing the number of slaves requested to cooperate at the same time. The greater Master Managing System is the Master Robot Managing System, followed soon after by the Master Environment Managing System. The first has the task to manage exceptional events such as airplanes' landings or departures: depending on the number of the expected pieces of luggage, it will awake a proper number of robots and then asleep them once their work will be accomplished. The second has the task to sort incoming messages from the environment to the correct topic.

Instead, there is not a limit to the number of slaves. Slaves are, in fact, robots. They can be scheduled and unscheduled at runtime, as far as they have a univocal ID. Through themselves, slaves have a *AllTheSame* policy as long as no conflicts occur. In case of unsolved conflicts, the robot with the minor ID will have a major priority. This allows the privileged robot not to stop its path in case of risk of collisions; also, with honours come responsibilities and in case of need it will be the first to be awaken and the last to send under charge (again: just in case of unresolved conflicts). Anyway, at anytime, priority is given to proximity to destination and/or battery level and then, in case of doubt, to ID number.

In the next sections it is shown how this happens, specifically.

6.2 MAPE-K Java Class

In this project, a standard implementation of a MAPE-K class is provided and can be found in listing 6.1. Depending on each MAPE-K Managing System, monitor(), analyze(), plan() and/or execute() methods are overridden.

```
1  public abstract class MAPEKclass {
2    public void callMAPE(){
3      monitor();
4      analyze();
5      plan();
6      execute();
7    }
8
9    public void monitor() {
10     //..do some computation to monitor system
11   }
12   public void analyze() {
13     //..do some computation to analyze system
14   }
15   public void plan() {
16     //..do some computation to plan actions
```

```
17    }
18    public void execute() {
19      //..do some computation to execute actions
20    }
21  }
```

Listing 6.1: MAPE-K java class

MAPE-K classes are encapsulated in robot components, see next section to understand how.

6.3 Robot Implementation

The main class of the project is the RobotImpl class. This class, as it extends Task and implements TaskInterface, is the wrapper containing all the systems needed to make a robot communicate with others, to make it perform repetitive actions and to check robots' status. This happens trough the use of already implemented and easy reusable Task's methods, relying on JMS to provide communication and on the java.util.concurrent library to manage periodicity. The idea is to have one dedicated process for each component. Moreover, every single robot instance will have one dedicated MAPE-K for each self-adaptive functionality needed.

As clarification, in this project, a component is intended as the module that perform actions and communicates periodically; instead, a robot is intended as the wrapper of a component and several MAPE-K classes.

The MAPE-K managing systems of the robots allow them to decompose and execute tasks: in fact, each robot is able to communicate in terms of high level tasks definition language (such as: *go to collect luggage*) and decompose them in subtasks iteratively (such as: *move wheels to go that direction*), according to the operation context down to the level of simple manipulation actions, which requires activating

the above capabilities concurrently.

In fig.6.1 it is possible to see a complete task process, from its birth (schedulation) to its death (unschedulation).

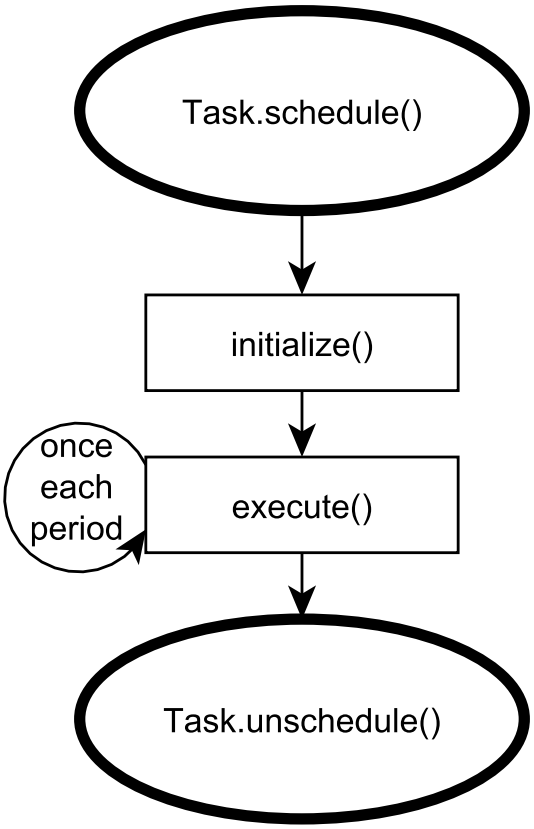

Figure 6.1: Activity diagram of a task process.

At first, each component has an initialization process in which it subscribes to pre-determined topics and becomes publisher on others. This happens in the *initialize()* method (see listing 6.2) which is run once before the first execution of the robot.

```
1 method initialize(){
2   Subscribe to and publish on RMSTopic;
3   Subscribe to and publish on RLMSTopic;
4   Subscribe to and publish on CSMSTopic;
5   Subscribe to and publish on ELMSTopic;
```

```
6     Subscribe to and publish on POSTopic;
7   }
```

Listing 6.2: method initialize

There are five general topics on which a robot component has to publish and to which it has to subscribe (see section 6.3.4 for further information). Through these topics all the management of robots' communication is achieved. Over communication, there are six fundamental skills that a robot has to present:

- Check calls from the master managing system to awake/asleep;
- Update active robots list;
- Check its integrity and call for help in case of a critical situation, then help other robots in critical situation;
- If not busy: check for new events such as new luggage;
- If moving: avoid possible collisions;
- If moving: manage a path.

All of these functionalities are invoked by java methods in the *execute()* method (see listing 6.3). In fact, each component runs autonomously once each period. So, once each period, the execute() method is called and all of these skills are autonomously put in operation at each execution of the robot. Periods are set in the .composite file associated with the SCA-XML definition of components. There is no need to synchronize robots' periods: in fact, each possible collision is managed through JMS communication. This is why each component can be scheduled and unscheduled at anytime, also at runtime, and each robot can have a different period.

```
method execute(){
  batteryDischarge();
  updateLists();
  RobotMS.callMAPE();
  Send a message to say that this robot is active;
  RobotListMS.callMAPE();
  CriticalSituationMS.callMAPE();
  EventLuggageMS.callMAPE();
  CollisionMS.callMAPE();
  PathFollowerMS.callMAPE();
  Send next position;
}
```

Listing 6.3: method execute

In fig.6.2 it is possible to see the execute method sequence, where the bold squares represent a MAPE-K class.

As the messaging system relies on JMS, each component may receive any message sent on a selected topic: this is also what underlies the decentralization of SAOMS. As far as urls are correctly set, each robot component may have a dedicate embedded software system, and master systems may be collocated on external work stations.

Master managing systems (such as MasterEnvironmentManagingSystem and MasterRobotManagingSystem) communicate new events and orders to robots sending messages on predetermined topics and then lose interest in them.

Robots receive messages and manage them updating lists and decomposing orders in lower tasks, each one managed by a dedicated onboard MAPE-K class.

RobotImpl provides several accessory methods to complete tasks, besides getters and setters.

In the next sections, alongside onboard dedicated MAPE-K classes presentation, some of these methods and functionalities will be shown and explained.

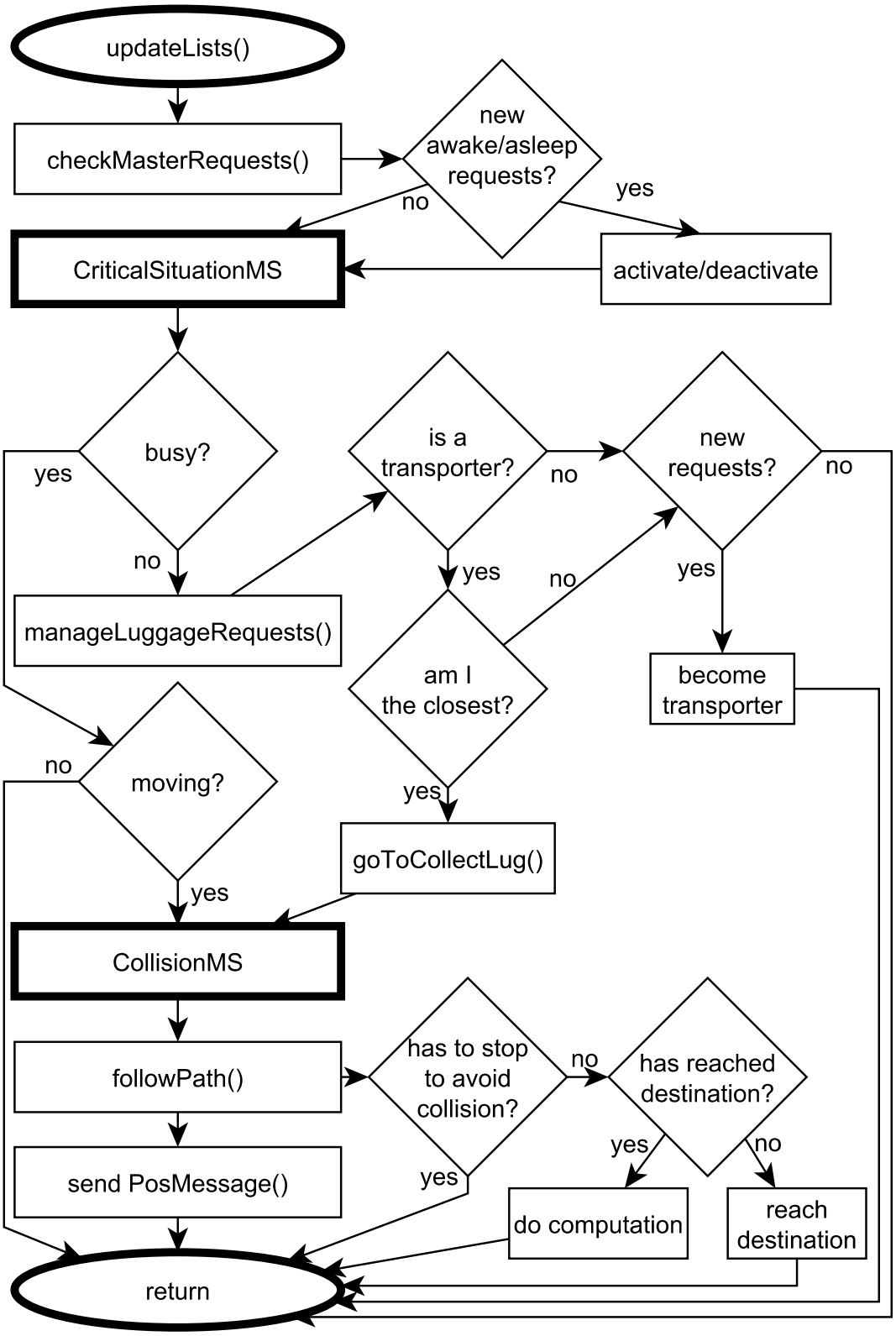

Figure 6.2: Activity diagram of the execute method.

6.3.1 Robot functionalities

Robots Lists Management

This section of implementation is included in *updateLists()* method and is used to update lists containing all available robots, luggage requests, other robots' positions and help requests. A robot sends a message once each period on RLMS topic to say it is active and a message at the end of its schedulation to say it will not be active anymore. In case of critical breakage of the communication unit on a robot, technical assistance is needed: someone else will send a message in place of the damaged robot.

Logic of these methods and functionalities can be found in fig.6.2.

Asleep/Awake Requests Management

This section of implementation is included in *checkMasterRequests()* method and is used to manage a asleep/awake requests. Requests are sent by MasterRobotManagingSystem (when flexibility is managed) and also by other robots (when critical situations occur, managing robustness).

Pseudocode of this method can be found in listings 6.4.

```
1  checkMasterRequests(){
2    Analyze Messages;
3    if(new messages){
4      if (this.robotID = Msg.ID){
5        if (Msg.ID = ACTIVE)
6          execution1;
7        if (Msg.ID = NOTACTIVE)
8          execution2;
9      }
10   }
11   if(execution1)
12     robot ready (not inCharge anymore);
13     if call by critical situation:
```

```
14        robot accomplishes other robot's task;
15    if(execution 2)
16      robot.setReadyToGoUnderCharge(true);
17  }
```

Listing 6.4: checkMasterRequests() method

During execution1, not only a robot will not be in charge anymore, but also, if activated because of a critical situation, it will emulate the other robot, which means that it will inherit the task of the injured. It is actually just a precaution because robots, even if asleep, when scheduled, receive all messages in every topic. Contrariwise, a new scheduled robot will start new and will not be able to heal robots in critical situation or to get old luggage. This may be useful just in the desperate case of several contemporary breakages.

When a robot is under charge its boolean variable *inCharge* is set to true and the robot is not available for new tasks. When it is awaken, this variable is set to false and the robot is ready to accomplish new jobs. Components present also another boolean variable *readyToGoUnderCharge* that is used to make the robot end the actual task and then go to under charge.

Luggage Requests Management

This section of implementation is included in *manageLuggageRequests()* and is used to manage luggage requests and responses. It receives luggage requests from the MasterEnvironmentManagingSystem and collect robots' responses. It is supposed that a Master request always comes before a response. If there are more than one response for the same luggage request, this method is the one which manages conflicts. The idea is that a robot gives a response one execution before starting its path to collect luggage (an auxiliary variable *isTransporter* is used). In the next

execution, after solving eventual conflicts, the robot is free to go (at this point the robot will become *busy*).

This method is called only when a robot is not busy and has enough battery.

Pseudocode of this method can be found in listings 6.5.

```
manageLuggageRequests(){
if(luggage request list is not empty)
  if(not already a transporter)
    take interest in the first luggage in the list;
  else
    get ready to check conflicts;
  if(luggage request list is not empty)
    if(newTransporter)
      remove request from luggage request list;
      send a message;
      become transporter;
  if(ready to check conflicts)
    check conflicts and eventually go;
}
```

Listing 6.5: manageLuggageRequests() method

Same policy and reasonings thath will be done for the checkConflictsAndGo() method are valid (see listing 6.9 for further details).

Path Following Management

This section of implementation is included in *followPath()* method and is used to make a robot move through a path to reach a destination, step by step.

Pseudocode of this method can be found in listings 6.6.

```
followPath(){
  if(robot has to stop)
    execution 0
  if(robot ready to go under charge)
    execution 5
  else{
    if(robot is in destination1)
```

```
8         execution 1;
9       if(robot is in destination2)
10        execution 2;
11      if(robot has a first destination){
12        execution 3;
13        destinationToReach=destination1;
14      }
15      if(robot has not a first destination){
16        execution 4:
17        destinationToReach=destination2;
18      }
19    }
20    switch(execution)
21      execution 0:
22        do nothing;
23      execution 1:
24        collect luggage;
25      execution 2:
26        deposit luggage;
27      execution 3 or 4:
28        reach destination;
29      execution 5:
30        reach chargePosition;
31  }
```

Listing 6.6: followPath() method

6.3.2 Onboard dedicated MAPE-K classes

There are two onboard dedicated MAPE-K classes for each robot and an additional one to manage the MasterRobotMS (MS is the acronym for Managing System). Here follows a brief description of these classes' functionality:

- CriticalSituationMS[Embedded in robot]: Check robot's integrity and call for help in case of a critical situation, then help other robots in critical situation;
- CollisionMS[Embedded in each robot]: If moving: avoid possible collisions;

- AwakeAsleepMS[Embedded in MasterRobot]: If needed: send a message to awake/asleep robots.

See next sections for further explanation.

Critical Situation Managing System

This is the class used to manage critical situation. A complete series of events is administered and in case of critical damage (such as breakage of the communication unit or of the movement actuators) a technical intervention is needed. Instead, whenever the critical situation do not avoid to the robot to go under charge and call other robots for help, there will be no need to communicate with the external evironment.

Pseudocode of this MAPE-K class can be found in listings 6.7.

```
1  MAPE-K class CriticalSituationMS{
2    callMAPE();
3    M: get criticalSituation variable;
4    A: analize criticalSituation variable;
5    P:
6     if (critic){
7      if (hasLuggage)
8        execution1: set to callForHelp();
9      if (NOT hasLuggage AND isGoingToCollectLuggage)
10       execution2: set to callForFinishPath();
11     if (NOT hasLuggage and NOT isBusy)
12       execution3: set to callSubstitute();
13     if (this robot was a healer)
14       sendForHelp=TRUE;
15     shutdown = TRUE;
16    }
17    if (NOT critic)
18     if (batteryLow)
19       execution4: get readyToGoUnderCharge();
20     else
21       execution5: get ready to collaborate;
```

```
22   E:
23    if (execution1 OR execution2 OR exection3){
24     send a message to ask for help with execution number;
25     ask human assistance;
26     if (sendForHelp = TRUE)
27       send a helpRequest for the Robot it was healing;
28    }
29    if(execution4){
30     set readyToGoUnderCharge(true);
31    }
32    if (shutdown = TRUE){
33     send a message to say the robot is not active anymore;
34     shutdownRobot;
35    }
36    if (execution5)
37     collaborateWithCSMS() to heal others;
38 }
```

Listing 6.7: CriticalSituationManagingSystem MAPE-K class

After checking critical situations on itself, each robot, if not already busy with another task, collaborates with CSMS. If another robot needs help, this robot will become a healer. The case of contemporary healing switches is rare but possible, due to physiological delays introduced by the transmission. This is why a robot becomes a healer first, and then, at the next execution, eventually go to heal. If two or more robots became healer of the same robot in a critical situation, the closest one goes. Here is where the NumberedSlave policy occurs: if two robots are at the same distance of the same selected destination, the one with the minor ID will be the actual healer.

In fig.6.3 it is possible to see the a graphic representation of the pseudocode, where the bold squares represent a MAPE-K class.

See more details in pseudocode in listing 6.8.

```
1 method collaborateWithCSMS() {
```

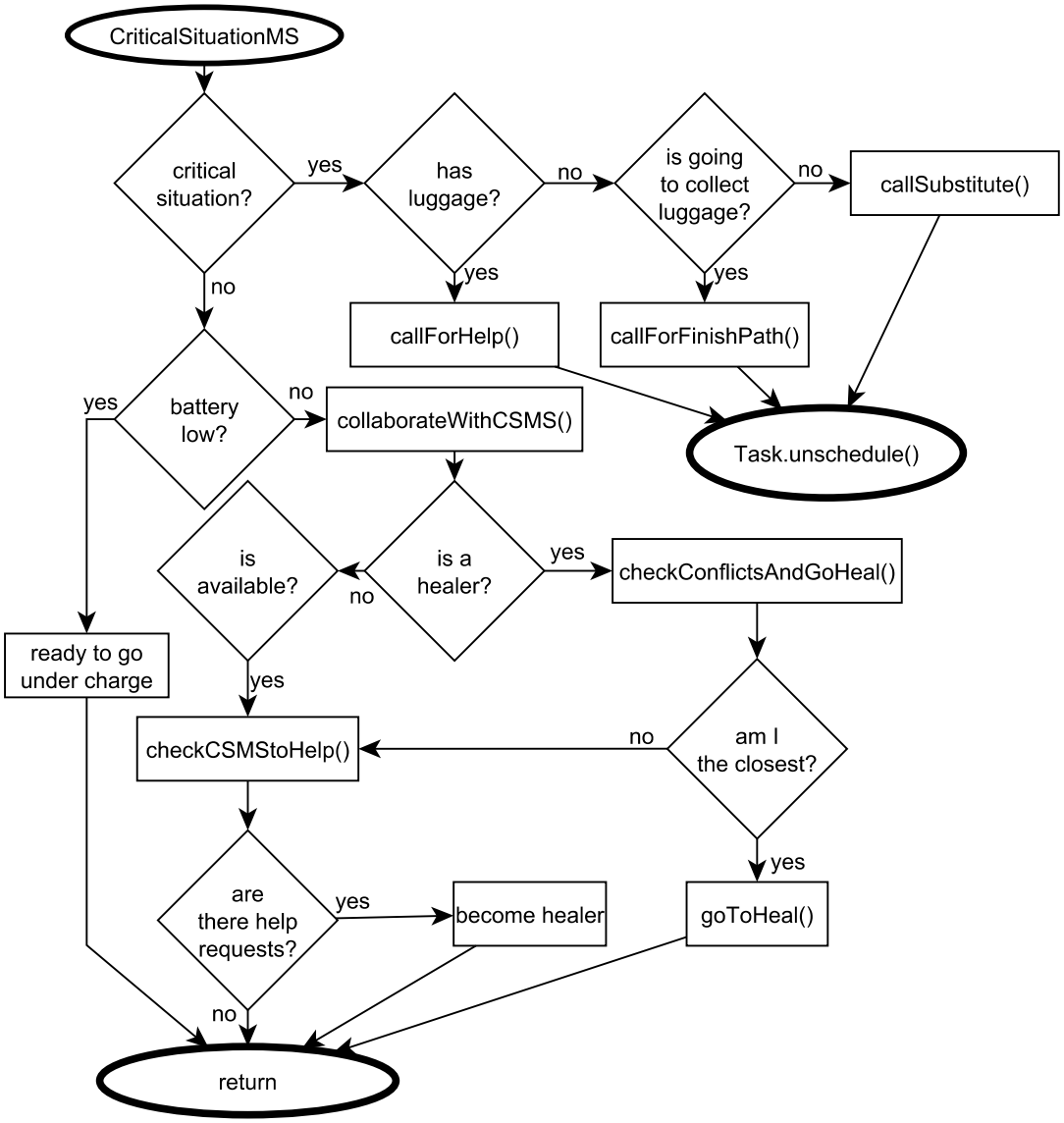

Figure 6.3: Activity diagram of CSMS.

```
2   if(robot is a healer)
3     checkConflictsAndGoHeal();
4   if(robot is available AND not in charge AND not a healer)
5     checkCSMStoHelp();
6 }
```

Listing 6.8: method collaborateWithCSMS

The *checkConflictsAndGo()* method (listing 6.9) comes around more than once (for example in the EventLuggageMS). The idea is always the same: first a robot declares its intention to go for a task, then, at the next execution, conflicts are checked and solved. This avoids that two or more robots fulfill the same task. If periods are correctly set dependently with the physical time needed to move wheels and other mechanical parts (messaging lapse of time is considered irrelevant), performance will not be significantly affected by this necessary double control.

```
1 methods checkConflictsAndGo{
2   Receive New Messages;
3   Update Request List;
4   if (Robots working on the same)
5     Save other robots' personal topics;
6   Get other robots positions;
7   if (amItheClosest())
8     go to heal;
9   Remove this request from the list;
10 }
```

Listing 6.9: method checkConflictsAndGo

If not already a healer (or not a healer anymore after checkConflictsAndGo() method), the robot will check for new and old help requests and eventually become a new healer (listing 6.10).

```
1 method checkCSMStoHelp(){
2   Receive new help requests and heal responses;
3   Update help request list;
4   If (helpRequesList is not empty){
```

```
5       Become healer;
6       Send message to say this robot became healer;
7     }
8   }
```

Listing 6.10: method checkCSMStoHelp

This allows a complete management of critical situations to improve robustness relying (as far as possible) just on slaves components.

Sensors and actuators of this MAPE-K class can be seen in fig.6.4.

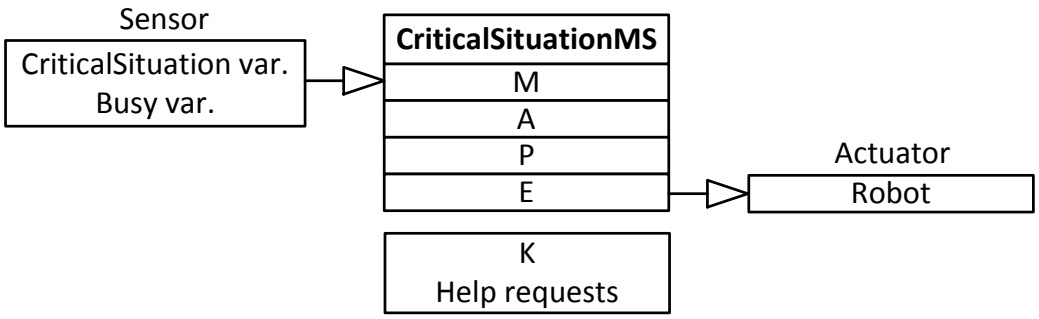

Figure 6.4: Sensors and actuators of CSMS.

Activity diagram of this MAPE-K class can be seen in fig.**??**.

Collision Managing System

This class manages robots to avoid eventual collisions. It receives other robots' position and set a boolean variable for the PathFollowerMS to eventually stop robots' path for one execution.

Pseudocode of this MAPE-K class can be found in listings 6.11.

```
1  MAPE-K class CollisionMS{
2    callMAPE();
3    M: Get other robots' positions;
4    A: Update other robots' positions list;
5       Analyze risk of collision;
```

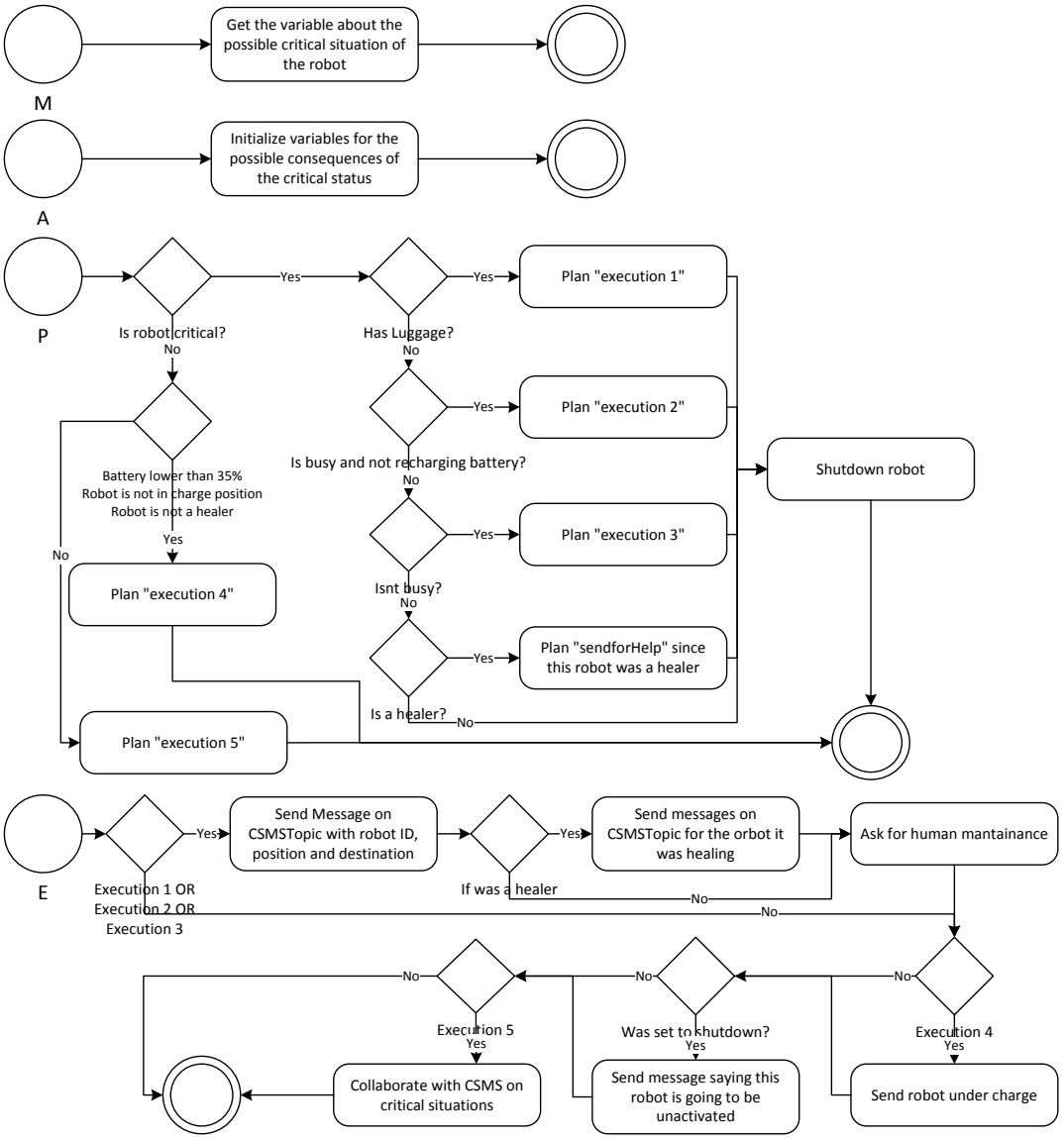

Figure 6.5: Activity diagram of CSMS.

```
6     P: if(risk of collision)
7          if(this robot id is not the minor){
8            get ready to stop one period;
9            exec=true;
10         }
11        else
12         exec=false;
13    E: if(exec)
14         setStopAPeriod(true);
15  }
```

Listing 6.11: CollisionManagingSystem MAPE-K class

Sensors and actuators of this MAPE-K class can be seen in fig.6.6.

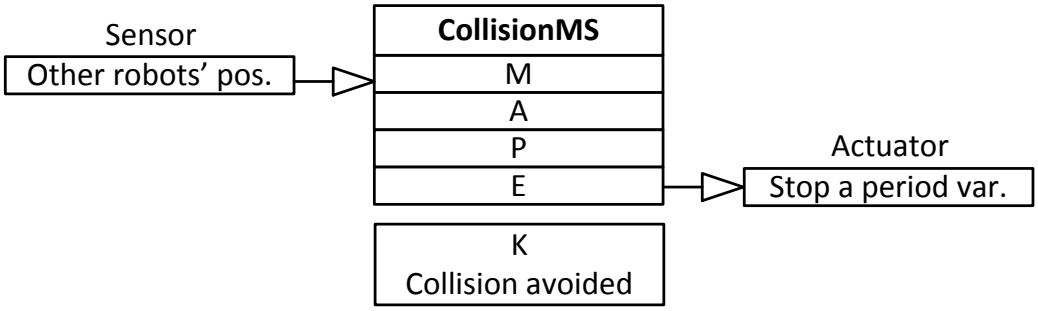

Figure 6.6: Sensors and actuators of CMS.

Activity diagram of this MAPE-K class can be seen in fig.6.7.

Awake Asleep Managing System

This MAPE-K class is a Master Managing System dedicated class, in fact it is included in the Master Robot Managing System java class. It allows to send messages on RobotMSTopic to awake/asleep robots, due to environment workload and robots' battery level.

Pseudocode of this MAPE-K class can be found in listings 6.12.

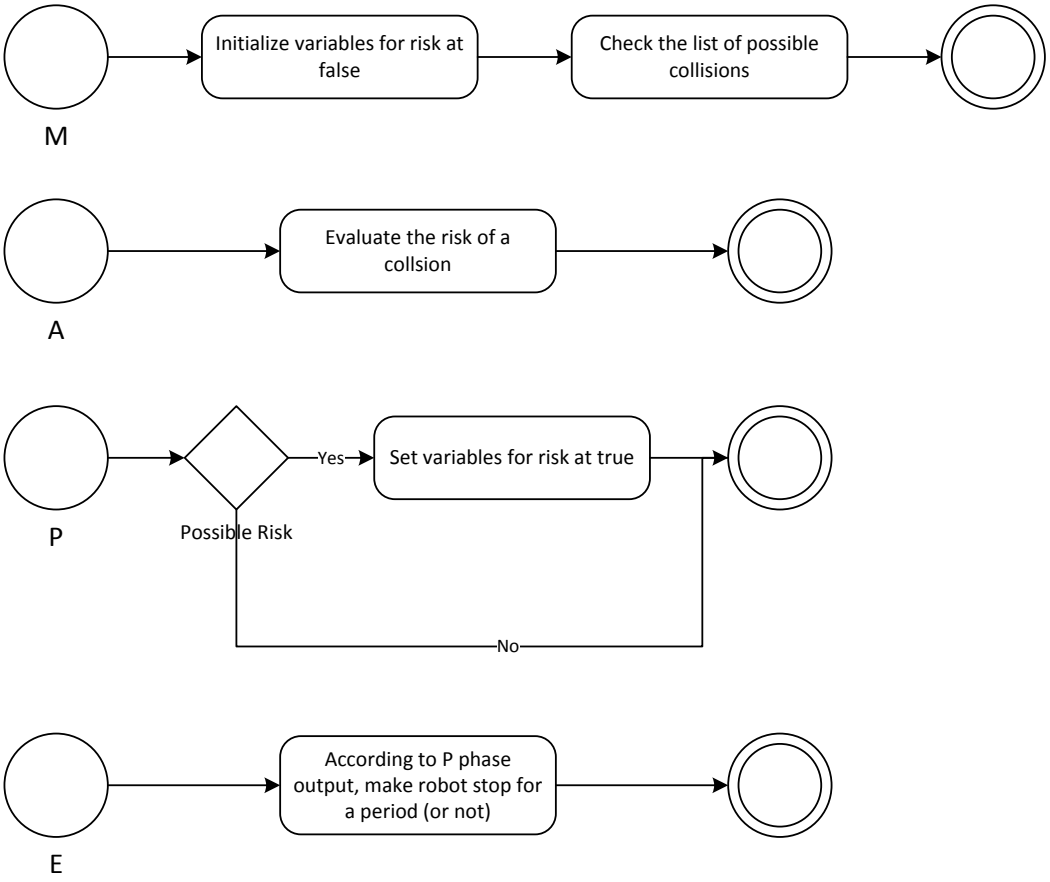

Figure 6.7: Activity diagram of CMS.

```
MAPE-K class AwakeAsleepMS{
  callMAPE();
  M: receive ExceptionalEventMsg;
  A: Analyze counter and newMessages;
  P: if(new robots to awake){
      update counter list;
      get ready to awake some robots;
     }
     if(counter came to an end)
      get ready to asleep some robots;
     Amount (awake-asleep) robots;
  E: if(amount!=0)
      awake/asleep robots;
}
```

Listing 6.12: AwakeAsleepManagingSystem MAPE-K class

Sensors and actuators of this MAPE-K class can be seen in fig.6.8.

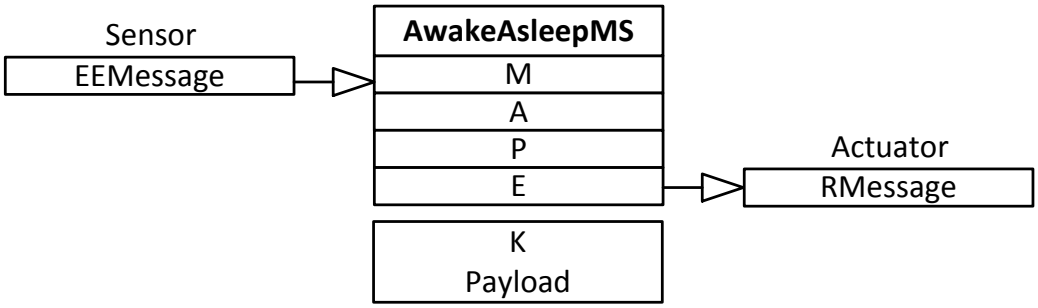

Figure 6.8: Sensors and actuators of AAMS.

Activity diagram of this MAPE-K class can be seen in fig.6.9.

Component Diagram

The component diagram of MAPE-K classes is depicted in fig.6.10.

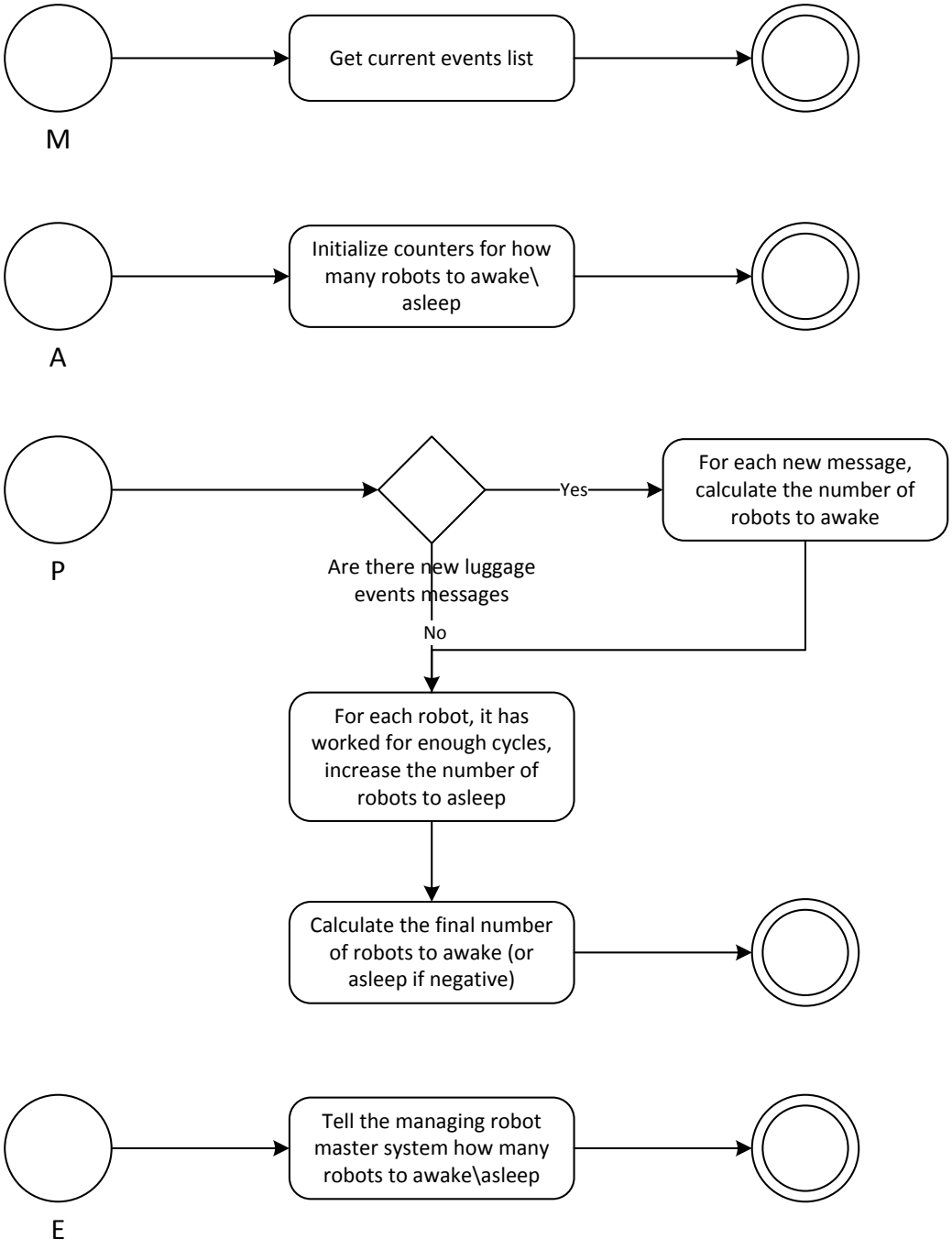

Figure 6.9: Activity diagram of AAMS.

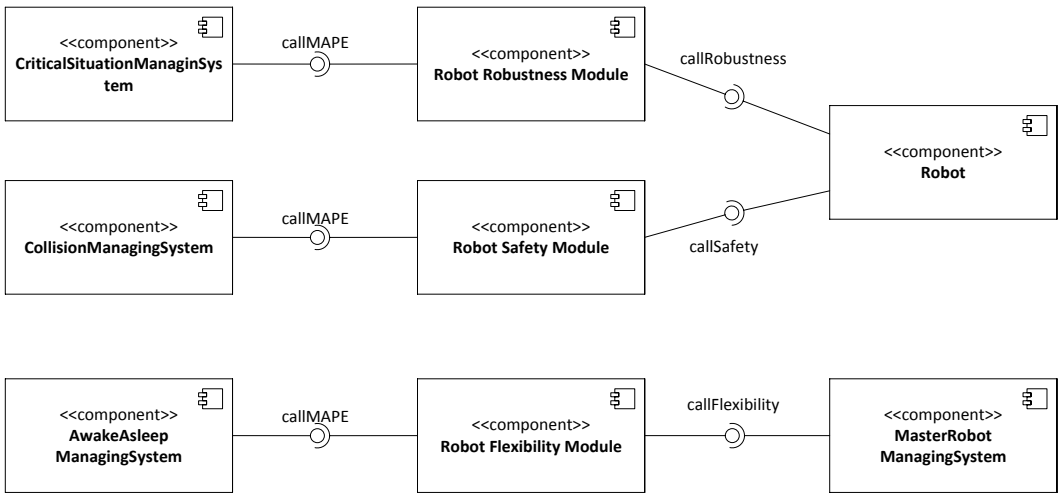

Figure 6.10: MAPE-K component diagram.

6.3.3 Further Considerations on Algorithms

The underlying algorithm is not quite simple. In this project, each event has been catalogued and managed to avoid errors.

6.3.4 Topics

There are six predetermined topics and one dedicated topic for each Robot component which is created along with robot's creation.

Robot MS Topic This is the topic used by the MasterRobotMS to communicate awake/asleep requests. It may be used also by components in case of critical situation to awake other robots. Each robot component receives periodically messages from this topic.

Robot List MS Topic This is the topic used by components to communicate which one is active and which one will not be active anymore. Each robot component and MasterRobotMS receive periodically messages from this topic.

Critical Situation MS Topic This is the topic used by components in case of critical situation to send help requests and heal responses. Each robot component receives periodically messages from this topic.

Event Luggage MS Topic This is the topic used by the MasterEnvironmentMS to communicate new luggage requests. It is used also by components to receive requests and send responses. Each robot component receives periodically messages from this topic.

Exceptional Event Topic This is the topic used by the MasterEnvironmentMS to communicate new exceptional events to the MasterRobotMS. MasterRobotMS receives messages from this topic.

Environment Topic This is the topic used by the Environment to communicate any kind of event. MasterEnvironmentMS receives messages from this topic.

Robot Position Topic This is the topic used by each component to communicate its position. Each robot component receives periodically messages from other robots' to avoid collisions.

What just explained can be seen in fig.6.11.

6.3.5 Messages

There are six different kind of message. Each one is represented by a java class implementing Serializable and providing a toString overridden method.

Robot Message This message is used to specify the ID of the robot wanted to awake/asleep. It can be sent by MasterRobotMS or, in case of critical situations, by

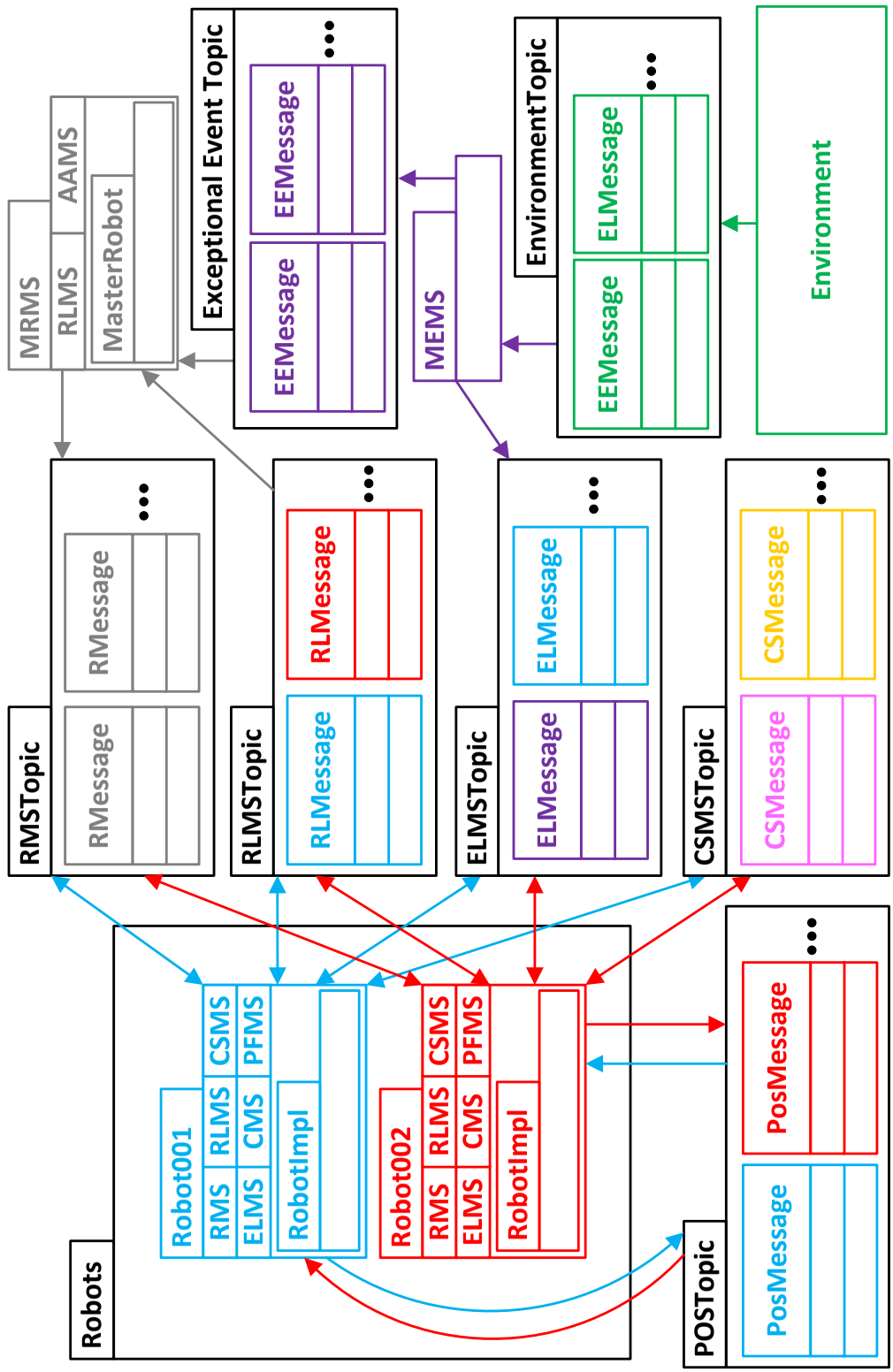

Figure 6.11: The use of topics by components.

other robots with an additional parameter *RobotToEmulate* to guarantee robustness: new awaken robots will receive old luggage and help requests lists.

UML of this class can be found in fig.6.12

Robot List Message This message is used to communicate that a robot is active or will not be active anymore. It is bounded to schedule() and uschedule() methods of the component. It can be sent by components and received by other components and the MasterRobotMS.

UML of this class can be found in fig.6.13

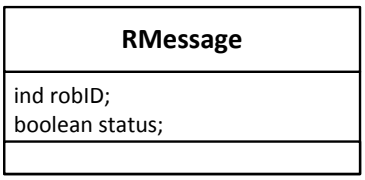

Figure 6.12: RMS msg.

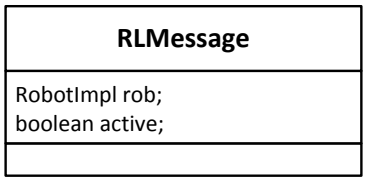

Figure 6.13: RLMS msg.

Critical Situation Message This message is used to communicate help requests and heal responses. It can be sent and received only by components.

UML of this class can be found in fig.6.14

Event Luggage Message This message is used to communicate new luggage transport requests and responses. It is used by components and the MasterEnvironmentMS.

UML of this class can be found in fig.6.15

Positon Message This message is used to communicate each robot's position at each execution. It is used only by components.

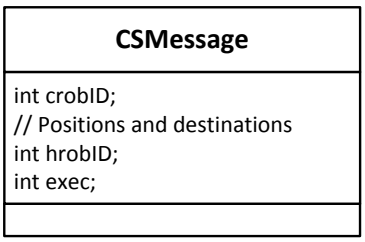

Figure 6.14: CSMS msg.

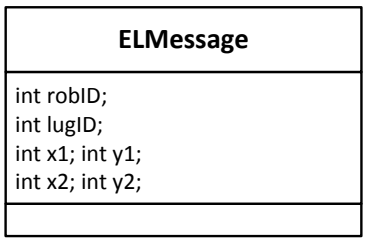

Figure 6.15: ELMS msg.

UML of this class can be found in fig.6.16

Exceptional Event Message This message is used to communicate exceptional events such as new landings or take off. It is used by the Master Managing Systems to communicate through themselves. It contains an estimated quantity of new luggage.

UML of this class can be found in fig.6.17

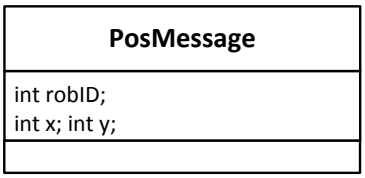

Figure 6.16: Position msg.

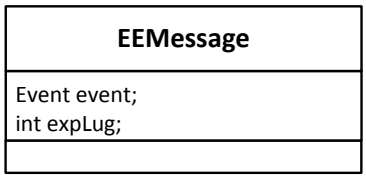

Figure 6.17: EEMS msg.

To end this section, figure 6.18 shows which Message is meant to be in which Topic.

6.3.6 unibg.saoms.util

There are four auxiliary classes to facilitate RobotImpl and MasterRobotMS.

BlockPos This class is used to divide space in blocks and identify them with x and y coordinates. It presents method toString and to set and get positions.

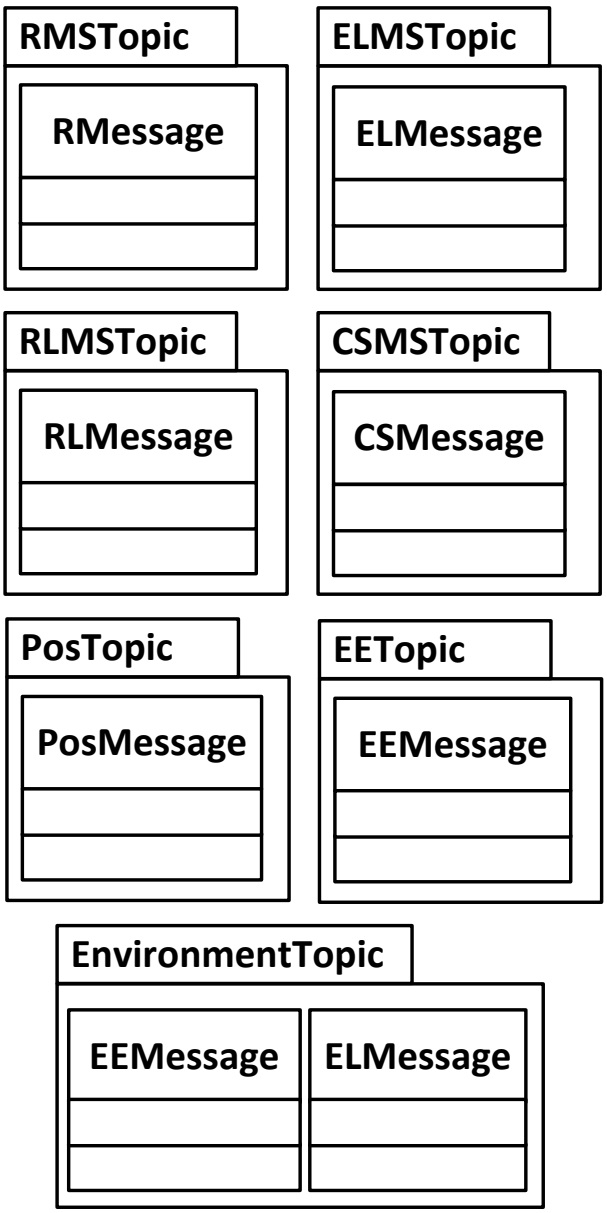

Figure 6.18: The use of topics by components.

RobotID This class is used to identify a robot. It provides equivocal name and number variables.

MRcounter This class is used to create a list of operating robots associated with a counter. Counters help MasterRobotMS to monitor robots' execution's durations.

Robot000 This class is used to define a fictitious robot to help MasterRobotMS and make it possible for it to benefit methods from the RobotImpl class.

6.3.7 Lower Level Features

In *collectLuggage()*, *depositLuggage()* and *reachBlock()* methods, *OUTcollectLuggage()*, *OUTdepositLuggage()* and *OUTmoveWheelsLuggage()* methods are located in the right place and that can be overridden by real execution algorithms (for example by ROS) to transform code in real movements.

In chapter 10 it is explained how to implements new functions such as, for example, getting images periodically from the camera.

6.3.8 Activity Diagram

Activity diagram of this class can be seen in fig.6.19.

6.4 Master Managing Systems

There are two main Master Managing Systems provided to manage robots and create a link from the external environment to the internal system:

- Master Environment Managing System;
- Master Robot Managing System.

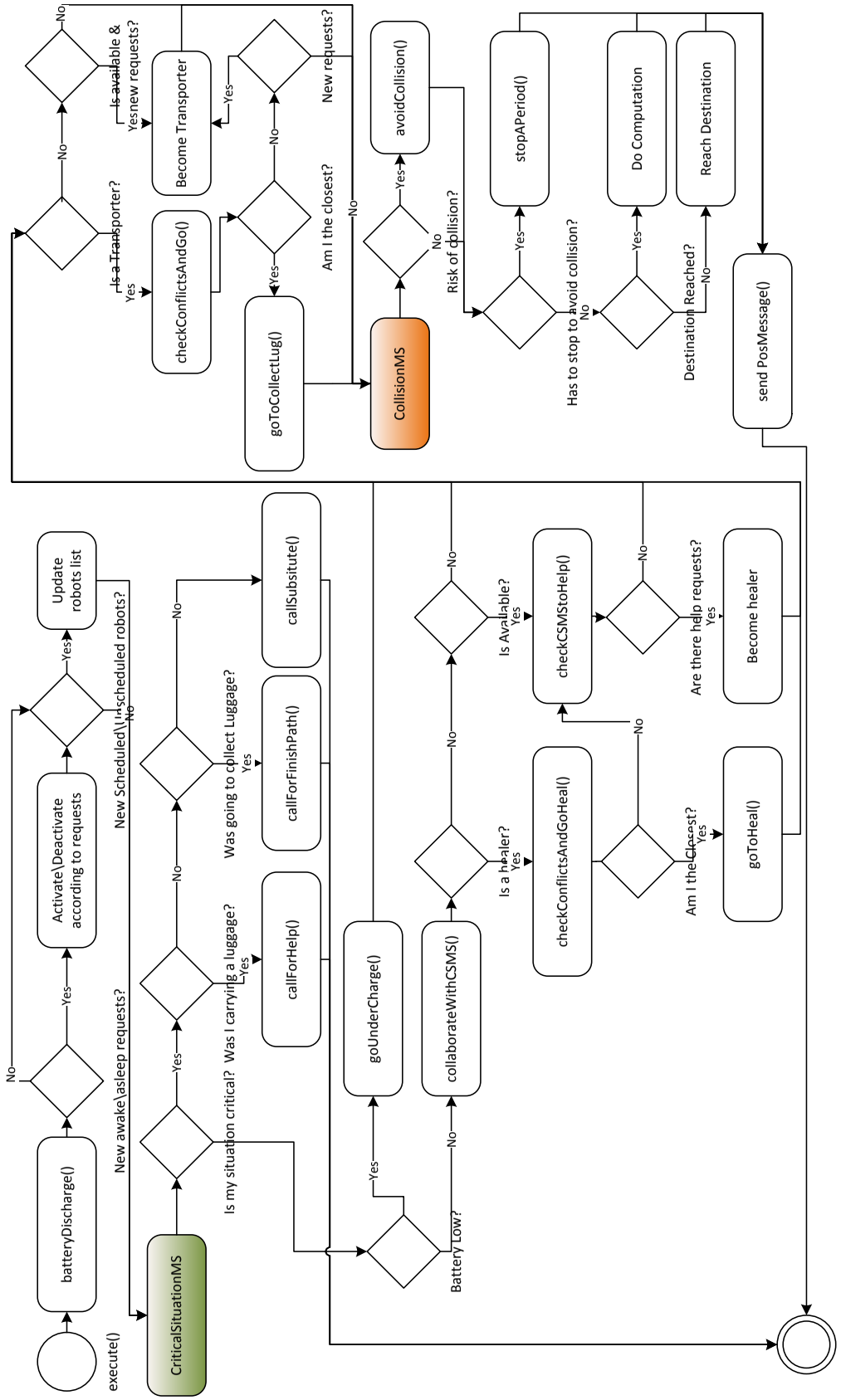

Figure 6.19: Activity diagram.

6.4.1 Master Environment Managing System

This Master Managing System is the access key to SAOMS from the external environment. It is implemented as a Java class that extends Task and Implements TaskInterface, in other words it executes periodically to check new messages from the enviroment. The communication channel to access this class is, once again, JMS. It is provided a Topic, named *EnvironmentTopic* on which messages such as "New airplane landing" or "New luggage to bring from A to B" are sent without distinction. The MasterEnvironmentMS will recognize messages, divide them by category and retrieve them to the proper Topic. For example, a message such as "New luggage to bring from A to B" will sent on EventLuggageMSTopic.

It does not present MAPE-K classes as its function is just to receive messages from the external environment, to categorize them and to forward them on the correct topic.

State of the art MasterEnvironmentManagingSystem is able to receive and recognize two kind of messages (see fig.6.15 and 6.17):

- EventLuggageMessage;
- ExceptionalEventMessage.

The first one is meant to be sent on EventLuggageMSTopic and the last one on ExceptionalEventTopic. Anyway, as well as the rest of this project, Master Environment Managing System class can be easily extended and through initialize() and execute() methods new functionalities can be seamlessly integrated. See listing 6.13 for a deepened example.

```
1 public class UserMasterEnvironmentMS
2                         extends MasterRobotManagingSystem{
3   // Fields
```

```
4     TopicObject userTopic = new TopicObject("USERTOPIC");
5
6     @Override
7     public void initialize{
8       super.initialize();
9       // Subscribe to and/or Publish on UserTopics
10    }
11
12    @Override
13    public void execute{
14      super.execute();
15      // Receive messages from UserTopics
16      // Do some computation
17      // Send messages to UserTopics
18    }
19  }
```

Listing 6.13: Example of MasterEnvironmentMS extension.

Activity diagram of this class can be seen in fig.6.20.

6.4.2 Master Robot Managing System

The last but not least Managing System and SAOMS class is the Master Robot Managing System. It is implemented as a java class containing dedicated MAPE-K classes. It can be seen as a middle way between a robot component and the Master Environment Managing System (see section 6.4.1). It provides some of the already implemented robot functionalities, such as activating asleeping robots. Moreover, it is geared with similar MasterEventMS' skills, such as receiving messages (*monitor*) and, after analyzing them (*analyze*), elaborating them (*plan*) and sending new messages to proper topics (*execute*). This is why MasterRobotMS is equipped with two dedicated MAPE-K classes: RobotListMS (see 6.3.1) which makes it similar to a robot component, and AwakeAsleepManagingSystem (see 6.3.2) which makes it a Master Managing System to all intents and purposes.

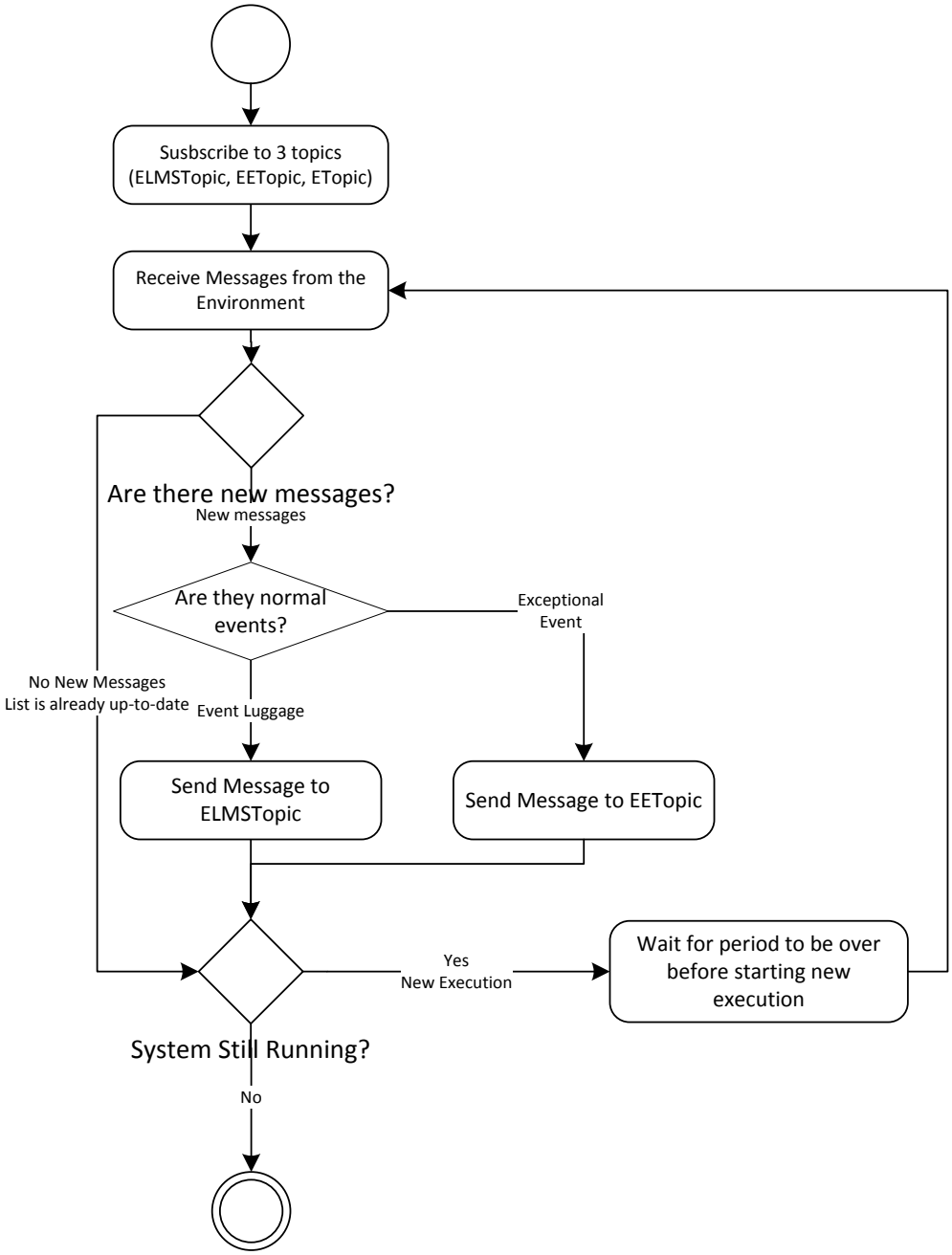

Figure 6.20: Activity diagram of MEMS.

Activity diagram of this class can be seen in fig.6.21.

6.5 UML and Class Diagram

As just explained there are three major classes and seven MAPE-K classes. In fig.6.22 all MAPE-K are shown just in terms of onboard dedicated classes, due to lack of white space on the paper. For further explanation and UML diagrams of these classes, please see section 6.3.2. A robot component can be seen as the wrapper of six MAPE-K onboard dedicated classes, of many protected fields and of several public methods (many more to those shown in the UML!), such as getters and setters. It relies on two exceptions: BatteryOutOfBoundException, in case of uncorrect battery level setting, and CriticalSituationException, in case of critical situation. Moreover, it uses two utility classes such BlockPos to collocate objects and itself in the frame of reference. In the end, an instance of this class, *masterRobot*, is used by the MasterRobotMS. This Master Managing class is the wrapper of two MAPE-K classes: RobotListMS, to update active robots list, and AwakeAsleepMS, to monitor new exceptional event, analyze them, plan and execute an action that could be to awake or asleep a robot, according to a dedicated MRcounter class, to establish if the robot has worked for enough time, and to the just updated list. ExceptionalEventMessages are sent by the MasterEnvironmentMS, after receiving it from the external environment. This last class is an instance of Task class.

6.6 Launchers

As this project has been developed to run also on different workstation concurrently, a server is needed. The message broker is the server and first item to launch, before launching the rest of components. To do so, two launcher java classes are provided.

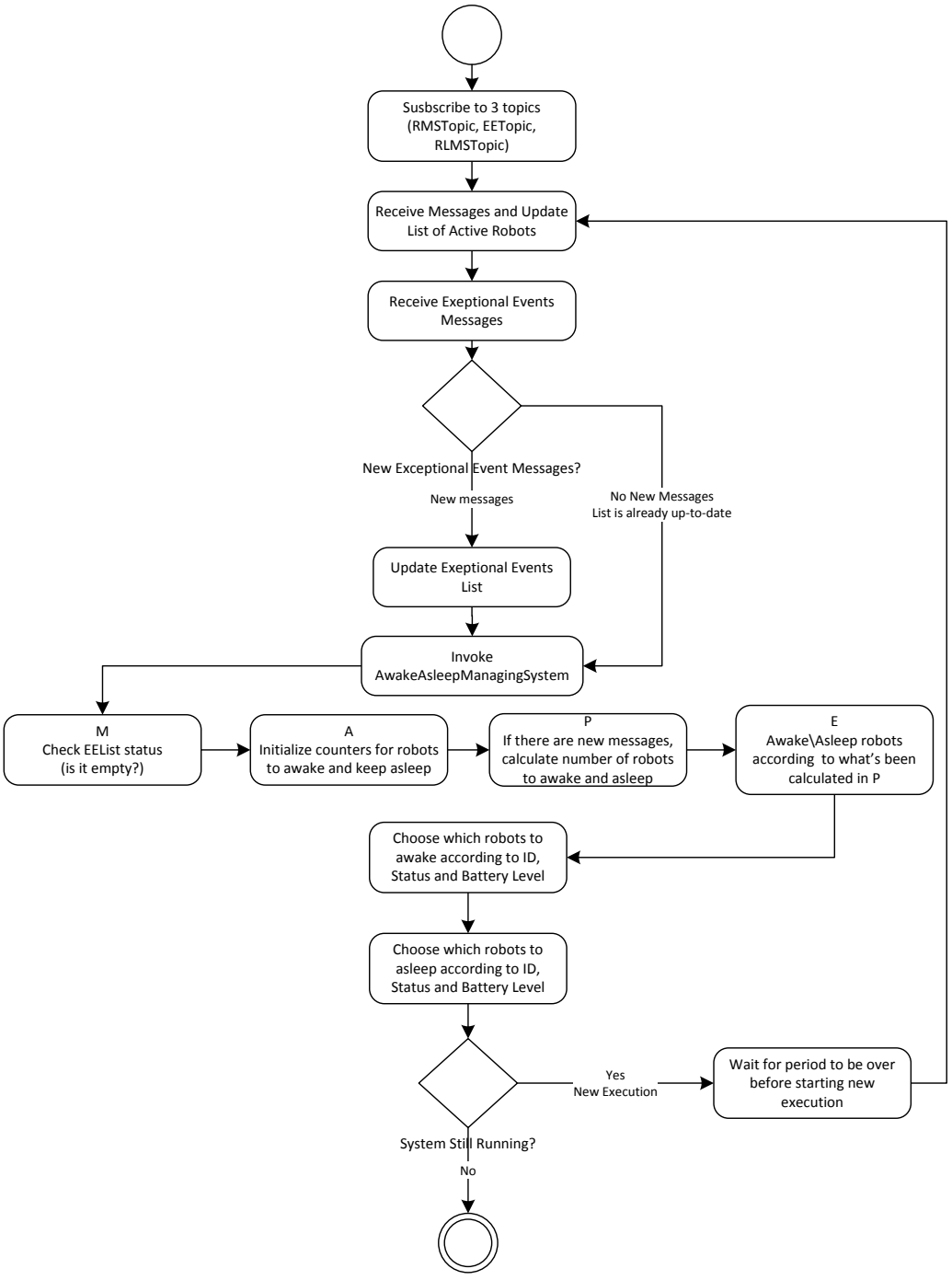

Figure 6.21: Activity diagram of MRMS.

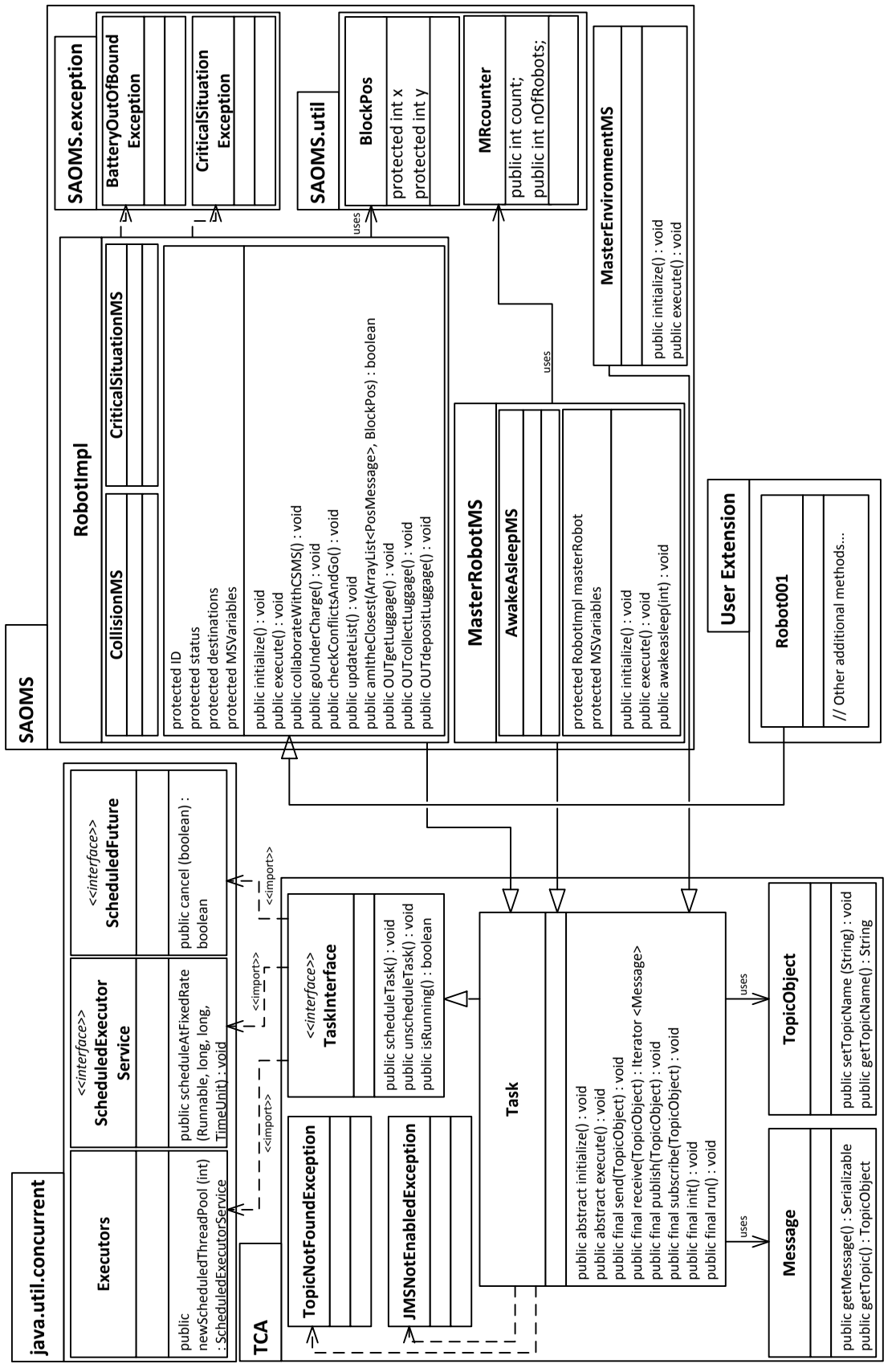

Figure 6.22: SAOMS Class Diagram.

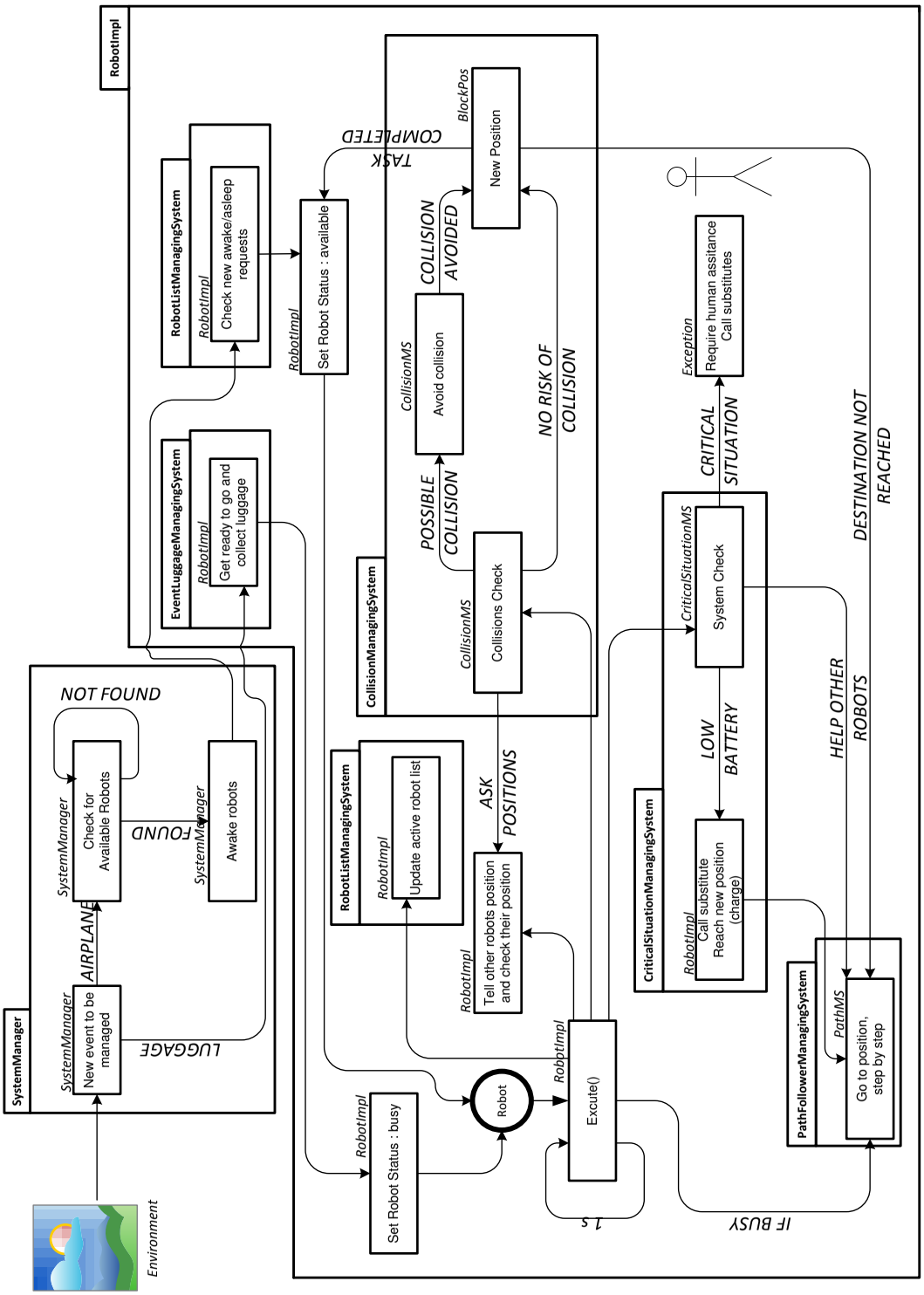

Figure 6.23: SAOMS UML.

MessageBroker Launcher creates a new instance of launchbroker.composite, the XML file containing the definition of the MessageBrokerComponent. See fig.6.24 to see a launcher java class example.

```
package launch;

import org.apache.tuscany.sca.host.embedded.SCADomain;

public class LaunchBroker extends Thread{
    public static void main(String[] args) throws Exception
    {
        LaunchBroker launch = new LaunchBroker();
        launch.start();
    }

    public void run()
    {
        System.out.println("Starting...");
        @SuppressWarnings("unused")
        SCADomain scaDomain = SCADomain.newInstance("launchbroker.composite");
    }
}
```

Figure 6.24: LaunchBroker java class.

Components Launcher creates a new instance of launchtasks.composite, the XML file containing the definition of all the new robot components the user wants to schedule. In case the application is already running, a new launcher java class and .composite file are needed.

In the next section it is explained how to define new components in the XML file.

6.6.1 XML

There are one XML file needed to be associated to the message broker and at least one for each new schedulation of robot components.

launchbroker.composite is standard-implemented and must not be modify by the user. Instead, if the user wants to use another server, *IP* (that can be seen in fig.6.25) has to be set with the URL of the selected server. In the end, one message broker has to be the only active JMS server for the whole execution of the program and must not to be stopped during the whole program execution.

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://launch" name="launchbroker">

    <component name="MessageBrokerComponent">
        <implementation.java
            class="unibg.robotics.tca.MessageBrokerImpl" />
        <property name="IP">localhost</property>
        <property name="port">61616</property>
        <service name="MessageBroker">
            <interface.java
                interface="unibg.robotics.tca.MessageBroker" />
        </service>
    </component>
</composite>
```

Figure 6.25: launchbroker.composite XML class.

launchtasks.composite contains the definition of one or more robot components. Properties of each component has to be set, and the same reasoing about the *IP* property can be done. The list of properties can be seen in fig.6.26.

- ID: is the univocal name of the component;
- period: is the period of execution of the component. It is suggested (but not necessary) to set a common period for all the robot components;
- delay: is the time a component has to wait to start its first execution once scheduled;

- runOnStartup: manages a boolean variable to establish if a robot has to run on startup or wait for a later call. For this project, it is suggested to make all robot components run on startup (unless in case of critical situations to be solved);

- mayInterruptIfRunning: manages a boolean variable to establish if a robot may interrupt its execution if running. The value of this property depends on level of criticality of the component: for example master components may not be interrupted while running, as the whole system depends on them;

- useJMS: is the property to setup JMS system and ports: obviously, for this project, it has necessarily to be set true;

- url: is the property to set the url of the message broker and its port.

- topicRename: provides more elasticity in naming topics: if correctly set (FirstTopicName::SecondTopicName), it allows to associate two different names to the same topic. It is mostly used to avoid errors due to case sensitive mistakes;

Further information can be found exploring TCA [2][3].

6.6.2 Woking on Different Workstations

As just explained, a URL property is provided to allow communication between different workstation.

To do so, it is enough to set the address of the MessageBroker in the launch-tasks.composite file, instead of the tag "localhost". Obviously, the components need to be linked through the same network.

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://launch"
    name="launchtasks">

    <component name="Robot001">
        <implementation.java class="robot.Robot001" />
        <property name="ID">Robot001</property>
        <property name="period">1000</property>
        <property name="delay">0</property>
        <property name="runOnStartup">true</property>
        <property name="mayInterruptIfRunning">true</property>
        <property name="useJMS">true</property>
        <property name="url">localhost:61616</property>
        <property name="topicRename">POS001,pos001::POS002,pos002</property>
        <service name="TaskInterface">
            <interface.java interface="unibg.robotics.tca.TaskInterface" />
        </service>
    </component>

    <component name="Robot002">
        <implementation.java class="robot.Robot002" />
        <property name="ID">Robot002</property>
        <property name="period">1000</property>
        <property name="delay">1000</property>
        <property name="runOnStartup">true</property>
        <property name="mayInterruptIfRunning">true</property>
        <property name="useJMS">true</property>
        <property name="url">localhost:61616</property>
        <property name="topicRename">POS001,pos001::POS002,pos002</property>
        <service name="TaskInterface">
            <interface.java interface="unibg.robotics.tca.TaskInterface" />
        </service>
    </component>
</composite>
```

Figure 6.26: lauchtasks.composite XML class.

6.7 Recap of variables

For a complete understanding of the project, it is useful to think about variables and their values depending on situations. In fig.6.27 it is possible to see their changing. Please, notice that the critical situation is not complained, as an external breakage needs to occur, and in that case it is not possible to know the effect of such a damage.

Situations	Boolean Variables								
	busy	transporter	hasLuggage	stopAPeriod	readyToGoUnderCharge	reachingCharge	inCharge	healer	critical
Awake in charge position	✗	✗	✗	✗	✗	✗	✗	✗	✗
Take interest in a piece of luggage	✗	✓	✗	✗	?	✗	✗	✗	✗
Starts its path to collect luggage	✓	✓	✓	✗	?	✗	✗	✗	✗
Risk of collision (no precedence)	✓	✓	✓	✓	?	?	✗	✗	✗
Low battery	?	?	?	?	✓	?	✗	✗	✗
Reaching charge position	✗	✗	✗	✗	✓	✓	✗	✗	✗
Reached charge and aslept	✗	✗	✗	✗	✗	✗	✓	✗	✗
Help other robots	✓	✗	✗	✗	✗	✗	✗	✓	✗

Figure 6.27: Variables changing.

Chapter 7

Use Cases

This project provides several pre-established use cases. Yet, thanks to its versatility, new use cases can be easily imaginable and implementable. Further information on user extension can be found in chapter 10.

Here is a list of classic use cases:

New luggage arriving

Brief Description A new piece of luggage arrives in a determined position.

Actors Environment, MasterEnvironmentMS, Robots.

Preconditions MasterEnvironmentMS is scheduled and executing. There is at least one scheduled Robot.

Trigger Environment sends a message on EnviromentTopic to advise about new luggage, its position and its destination.

Postconditions Luggage has been brought from one place to another.

Standard process

- 1) Environment sends a message on EnviromentTopic to advise about new luggage, its position and its destination;
- 2) MasterEnvironmentMS receives new messages from EnvironmentTopic;
- 3) MasterEnvironmentMS recognizes a new message about luggage;
- 4) MasterEnvironmentMS forwards the message on EventLuggageMSTopic;
- 5) Available Robots receive new messages from EventLuggageMSTopic;
- 6a) One available Robot takes care of luggage, becomes busy and sends a message on EventLuggageMSTopic;
- 6b) More than one available Robots take simultaneously care of the same luggage, become busy and send the same messages on EventLuggageMSTopic;
- 7a) Robot starts its path to go and collect luggage;
- 7b.a) The closest robot starts its path to go and collect contested luggage;
- 7b.b) Other robots become available again;
- 8) Robot reaches luggage and collect it;
- 9) Robot starts its path to go and bring luggage to destination;
- 10) Robot reaches destination and deposit luggages;
- 11) Robot becomes available again;

Airplane take-off/landing

Brief Description A new plane takes-off/lands in the airport.

Actors Environment, MasterEnvironmentMS, MasterRobotMS, Robot.

Preconditions MasterEnvironmentMS and MasterRobotMS are scheduled and executing.

Trigger Environment sends a message on EnviromentTopic to advise about new plane and expected luggage.

Postconditions Due to expected luggage, a certain number of Robots are activated.

Standard process

- 1) Environment sends a message on EnviromentTopic to advise about new plane and expected luggage.
- 2) MasterEnvironmentMS receives new messages from EnvironmentTopic;
- 3) MasterEnvironmentMS recognizes a new message about an exceptional event;
- 4) MasterEnvironmentMS forwards the message on ExceptionalEventTopic;
- 5) MasterRobotMS receives new messages on ExceptionalEventTopic;
- 6) MasterRobotMS analyzes new message;

- 7) MasterRobotMS analyzes its counter to check if old workload has been disposed;
- 8) MasterRobotMS analyzes which Robot is better awake/asleep, due to battery level and ID number;
- 9) Due to expected luggage, already disposed workload and priority MasterRobotMS establishes how many and which Robots awake or asleep;
- 10) MasterRobotMS sends several messages on RobotMSTopic to awake or asleep a corresponding number of Robots;
- 11) Robots receive new messages from RobotMSTopic;
- 12a) Awaken Robots may get ready to asleep as soon as finished their task, depending on their battery level and ID number;
- 12b) Aslept Robots may awake, depending on their battery level and ID number;
- 13a) Chosen Robot finishes its path and go under charge;
- 13b) Chosen Robot gets ready to accomplish new tasks.

Similar considerations can be done for an airplane taking off.

Robot critical situation

Brief Description A Robot occurs in a critical situation.

Actors Robots.

Preconditions There is at least one scheduled Robot.

Trigger Robot's critical situation boolean variable is set to true by hardware.

Postconditions Robot has a substitute and, if too critical, asks for assistance.

Standard process

- 1) Robot's critical situation boolean variable is set to true by hardware;
- 2) Robot sends a message on CriticalSituationMSTopic explaining its condition to ask for help;
- 3) Available Robots collaborate with CriticalSituationMS and receive messages from CriticalSituationMSTopic;
- 4a) An available Robot finds the new message and becomes healer;
- 4b) More than one available Robot simoultaneously find new messages and become healers;
- 5a) Robot starts its path to help the Robot in the critical situation;
- 5b.a) The closest Robot starts its path to help the Robot in the critical situation;
- 5b.b) Other Robots start collaborating with CriticalSituationMS once again;

- 6a) Healer Robot calls a substitute to the broken Robot if it was not following a path;
- 6b) Healer Robot calls a substitute and follows the broken Robot path, if it was following a path;
- 6c) Healer Robot calls a substitute and go to collect luggage from the broken Robot, if it already collect luggage;
- 7) Damaged Robot asks for assistance.

Robot awaking

Brief Description A Robot receives an awake message and becomes available.

Actors Robots.

Preconditions Robots are scheduled but not available.

Trigger New Message of awakening on RobotMSTopic.

Postconditions One Robot becomes available.

Standard process

- 1) Not available Robot checks RobotMSTopic for new messages;
- 2) Not available Robot finds awakening message on RobotMSTopic with its ID;
- 3) Not available Robot becomes available.

Chapter 8

Testing and Validation

The best way to test this project's features is to create different scenarios and check the correct operation of the algorithms. The idea is to test the environment under normal and stressful or critical conditions.

8.1 Testing one Robot with several pieces of luggage

Src code enclosed under the name of *SAOMS_test01*.

As already explained, the components are instantiated (see fig.8.1). Then, one execution for each period of each component occurs. This fact is highlighted on the console through a line with a time stamp indication. Please, notice that each execution starts with the precision of a hundredth of a second.

Then, the Environment warns about the arrival of a new piece of luggage. As explained, the MasterEnvironmentManagingSystem receives and manages the new event and retrieves the message on the proper Topic. Robot001, as the only active robot, is the one which takes care of the new request thanks to its MAPE-K EventLuggageManagingSystem (see in fig.8.2).

Even if the Environment and the MasterEnvironmentManagingSystem highlight

```
Starting...
gen 23, 2015 2:32:33 PM org.apache.tuscany.sca.node.impl.NodeImpl <init>
Informazioni: Creating node: launchtasks.composite
gen 23, 2015 2:32:33 PM org.apache.tuscany.sca.node.impl.NodeImpl configureNode
Informazioni: Loading contribution: file:/C:/Users/Yamuna/UNI/workspace/INFO3B/SAOMS_test01/bin/
gen 23, 2015 2:32:34 PM org.apache.tuscany.sca.node.impl.NodeImpl start
Informazioni: Starting node: launchtasks.composite
- Successfully connected to tcp://localhost:61616
MasterEnviromentManagingSystem starts at 2015-01-23 14:32:35.274
- Successfully connected to tcp://localhost:61616
MasterRobotManagingSystem starts at 2015-01-23 14:32:35.316
- Successfully connected to tcp://localhost:61616
Enviroment tests start at 2015-01-23 14:32:35.328
Enviroment: ready to test at 2015-01-23 14:32:35.329
- Successfully connected to tcp://localhost:61616
Initialization of Robot001
Execution 0 of Robot001 starts at 2015-01-23 14:32:35.34
```

Figure 8.1: Initialization of the components.

```
Enviroment: Test Luggage 1: Luggage to bring from (3,1) to (3,10) at 2015-01-23 14:57:45.769
Execution 6 of Robot001 starts at 2015-01-23 14:57:48.8
Robot001 updated active robot list:             []
Robot001 help request list:                     []
Robot001 luggage request list:                  []
Robot001 battery level: 100.0
MasterEnviromentManagingSystem: new event received and managed at 2015-01-23 14:57:49.268
Execution 7 of Robot001 starts at 2015-01-23 14:57:49.809
Robot001 updated active robot list:             []
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (3,1) to (3,10)]
Robot1 is bringing luggage from (3,1) to (3,10)
```

Figure 8.2: Arrival of a piece of luggage.

the arrival of new luggage, Robot001 is already busy and does not take interest in it (see fig.8.3).

```
Enviroment: Test Luggage 2: Luggage to bring from (1,5) to (10,5) at 2015-01-23 14:57:48.784
Execution 7 of Robot001 starts at 2015-01-23 14:57:49.809
Robot001 updated active robot list:             []
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (3,1) to (3,10)]
Robot1 is bringing luggage from (3,1) to (3,10)
Robot001 battery level: 100.0
MasterEnviromentManagingSystem: new event received and managed at 2015-01-23 14:57:50.278
Enviroment: Test Luggage 3: Luggage to bring                    at 2015-01-23 14:57:49.785
MasterEnviromentManagingSystem: new event received and managed at 2015-01-23 14:57:50.78
Execution 8 of Robot001 starts at 2015-01-23 14:57:50.811
Robot001 updated active robot list:             []
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (1,5) to (10,5),
                                                 Luggage to bring from (10,8) to (1,8)]
```

Figure 8.3: Arrival of new pieces of luggage: Robot001 not interested.

Then, Robot001 follows its path to destination1 and collect the luggage (see fig.8.4).

```
Execution 8 of Robot001 starts at 2015-01-23 14:57:50.811
Robot001 updated active robot list:             []
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (1,5) to (10,5), Luggage
Robot001 says: My position now is: (0,1) and my next position will be: (1,1)
Robot001 battery level: 100.0
Execution 9 of Robot001 starts at 2015-01-23 14:57:51.803
Robot001 updated active robot list:             []
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (1,5) to (10,5), Luggage
Robot001 says: My position now is: (1,1) and my next position will be: (2,1)
Robot001 battery level: 100.0
Execution 10 of Robot001 starts at 2015-01-23 14:57:52.812
Robot001 updated active robot list:             []
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (1,5) to (10,5), Luggage
Robot001 says: My position now is: (2,1) and my next position will be: (3,1)
```

Figure 8.4: Robot001 follows its path to destination.

Only when it arrives to destination2 and deposit its luggage, it takes interest in the new luggage, as no other robots managed it: the luggage request list is updated (see fig.8.5). The Robot will take care of the first piece luggage arrived in the environment (chronological order).

This is the normal functionality of the program. However, after a while, Robot001

```
Robot001 says: I've arrived to destination.
Robot001 battery level: 74.0
Execution 23 of Robot001 starts at 2015-01-23 14:58:05.808
Robot001 updated active robot list:             []
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (1,5) to (10,5), Luggage
Robot001 says: Ready to deposit luggage in position (3,10)
Robot001 says: Luggage deposited.
Robot001 battery level: 72.0
Execution 24 of Robot001 starts at 2015-01-23 14:58:06.801
Robot001 updated active robot list:             []
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (1,5) to (10,5), Luggage
Robot1 is bringing luggage from (1,5) to (10,5)
Robot001 battery level: 70.0
Execution 25 of Robot001 starts at 2015-01-23 14:58:07.808
Robot001 updated active robot list:             []
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (10,8) to (1,8)]
```

Figure 8.5: Robot001 finishes its first job and starts a second one.

has low battery and, instead of taking care of the last piece of luggage, goes under charge on its own (see fig.8.6). Please, notice that the robot keeps its luggage list and send a message to ask for help, in particular, to call a substitute.

```
Robot001 has low battery and is going under charge...
Robot001 says: My position now is: (1,1) and my next position will be: (0,1)
Robot001 needs to reach charge position as soon as possible!
Execution 59 of Robot001 starts at 2015-01-23 14:58:41.808
Robot001 updated active robot list:             []
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (10,8) to (1,8)]
Robot001 has low battery and is going under charge...
Robot001 says: I've arrived to charge position.
Execution 60 of Robot001 starts at 2015-01-23 14:58:42.797
Robot001 updated active robot list:             []
Robot001 help request list:                     [New help request from Robot1]
Robot001 luggage request list:                  [Luggage to bring from (10,8) to (1,8)]
```

Figure 8.6: Robot001 goes under charge.

As no other robot are present, the help call is kept by Robot001 itself and itself will be the one who will accomplish the request once it will be fully charged. In fig.8.7 it is possible to see how the help request list is updated as well as the luggage list: in fact, the robot starts again its path to go and collect luggage that has not been collected yet.

This process is repeated continuously. No bugs and errors to signal during this

```
Execution 69 of Robot001 starts at 2015-01-23 14:58:51.803
Robot001 updated active robot list:             []
Robot001 help request list:                     [New help request from Robot1]
Robot001 luggage request list:                  [Luggage to bring from (10,8) to (1,8)]
Robot001 is now fully charged and ready (but still asleep)
Robot1 is a new healer for Robot1
Robot001 battery level: 100.0
Execution 70 of Robot001 starts at 2015-01-23 14:58:52.805
Robot001 updated active robot list:             []
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (10,8) to (1,8)]
Robot001 is now fully charged and ready (but still asleep)
Robot001 is a healer and now is awake
Robot1 is bringing luggage from (10,8) to (1,8)
```

Figure 8.7: Robot001 is fully charged and goes back to work.

test.

The complete robot's path can be seen in fig.8.8. Please, notice that robots can manage luggage also in unexpected positions.

8.2 Testing two Robots with several pieces of luggage

Src code enclosed under the name of *SAOMS_test02*.

As already explained, the components are instantiated (see fig.8.9). Then, one execution for each period of each component occurs. This fact is highlighted on the console through a line with a time stamp indication. Please, notice that each execution starts with the precision of a hundredth of a second.

Robot001 is the first to start, so it does not reveal the presence of other robots. Robot002 is the second one to begin its execution and, as the task was anyway scheduled at the start, it receives the message from Robot001 and updates its active robots' list. In execution 1, also Robot001 observes the message sent during execution 0 by Robot002.

Then, the Environment warns about the arrival of a new piece of luggage.

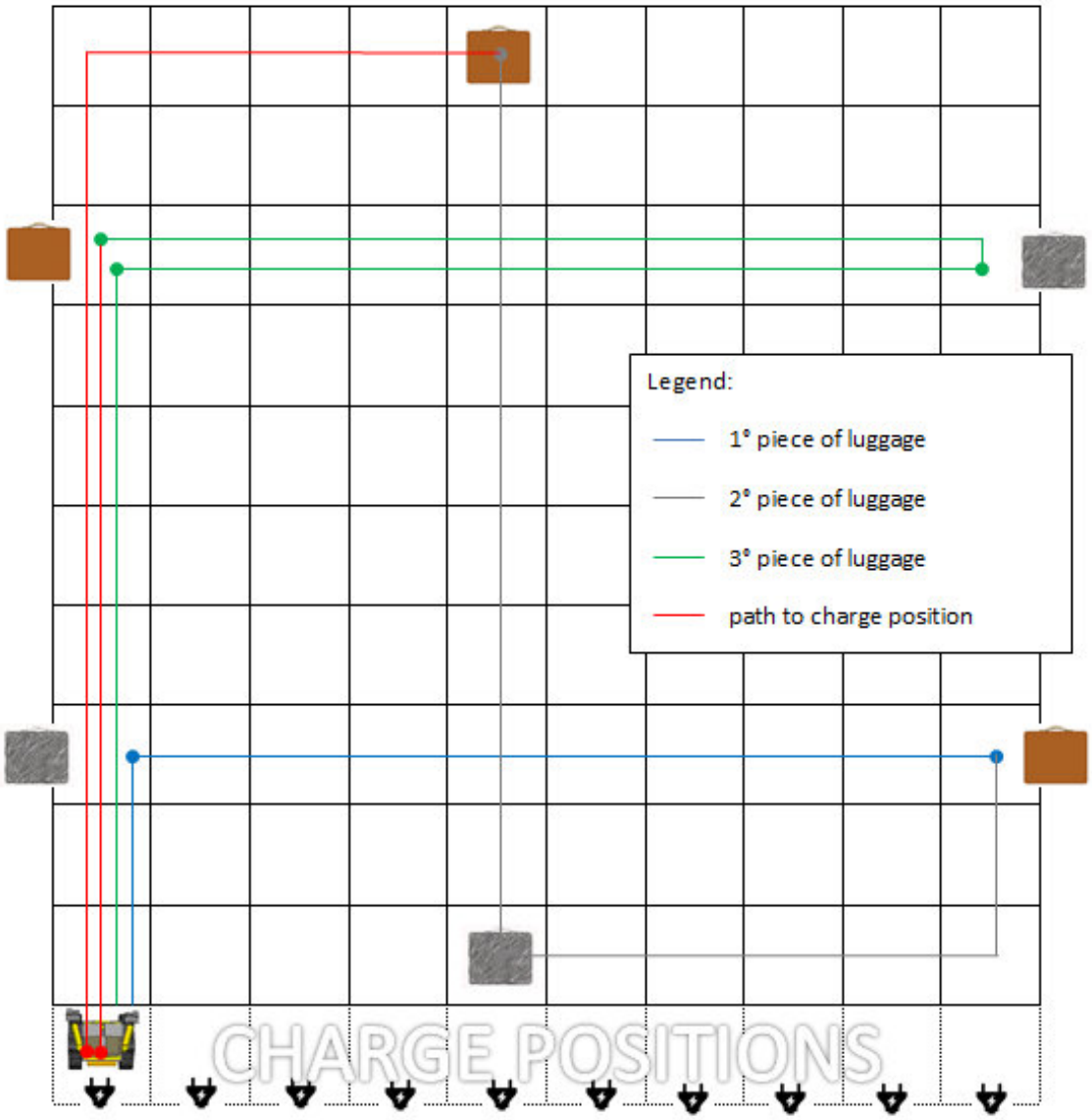

Figure 8.8: Robot001's complete path.

```
Starting...
gen 23, 2015 3:53:44 PM org.apache.tuscany.sca.node.impl.NodeImpl <init>
Informazioni: Creating node: launchtasks.composite
gen 23, 2015 3:53:45 PM org.apache.tuscany.sca.node.impl.NodeImpl configureNode
Informazioni: Loading contribution: file:/C:/Users/Yamuna/UNI/workspace/INFO3B/SAOMS_test02/bin/
gen 23, 2015 3:53:46 PM org.apache.tuscany.sca.node.impl.NodeImpl start
Informazioni: Starting node: launchtasks.composite
- Successfully connected to tcp://localhost:61616
MasterEnviromentManagingSystem starts at 2015-01-23 15:53:46.723
- Successfully connected to tcp://localhost:61616
MasterRobotManagingSystem starts at 2015-01-23 15:53:46.754
- Successfully connected to tcp://localhost:61616
Enviroment tests start at 2015-01-23 15:53:46.754
Enviroment: ready to test at 2015-01-23 15:53:46.754
- Successfully connected to tcp://localhost:61616
Initialization of Robot001
Execution 0 of Robot001 starts at 2015-01-23 15:53:46.77
Robot001 updated active robot list:             []
Robot001 help request list:                     []
Robot001 luggage request list:                  []
Robot001 battery level: 100.0
- Successfully connected to tcp://localhost:61616
Initialization of Robot002
Execution 0 of Robot002 starts at 2015-01-23 15:53:47.303
Robot002 updated active robot list:             [Robot1 is active]
Robot002 help request list:                     []
Robot002 luggage request list:                  []
Robot002 battery level: 100.0
Execution 1 of Robot001 starts at 2015-01-23 15:53:47.788
Robot001 updated active robot list:             [Robot2 is active]
Robot001 help request list:                     []
Robot001 luggage request list:                  []
Robot001 battery level: 100.0
```

Figure 8.9: Initialization of the components.

As explained, the MasterEnvironmentManagingSystem receives and manages the new event and retrieves the message on the proper Topic. It is possible to see in fig.8.10 how Robot001 executes an instant before the message is retrieved and so the Robot002 is the one which takes care of the new request thanks to its MAPE-K EventLuggageManagingSystem.

```
Enviroment: Test1: Luggage to bring from (3,1) to (3,10) at 2015-01-23 15:53:49.777
Execution 6 of Robot001 starts at 2015-01-23 15:53:52.783
Robot001 updated active robot list:             [Robot2 is active]
Robot001 help request list:                     []
Robot001 luggage request list:                  []
Robot001 battery level: 100.0
MasterEnviromentManagingSystem: new event received and managed at 2015-01-23 15:53:53.258
Execution 6 of Robot002 starts at 2015-01-23 15:53:53.297
Robot002 updated active robot list:             [Robot1 is active]
Robot002 help request list:                     []
Robot002 luggage request list:                  [Luggage to bring from (3,1) to (3,10)]
Robot2 is bringing luggage from (3,1) to (3,10)
```

Figure 8.10: Arrival of a piece of luggage.

The role of the Environment and of the MasterEnvironmentManagingSystem ends after the simulation of the arrival of other three pieces of luggage, almost at the same time. Robot002 is busy and does not take an interest in them, instead Robot001 updates the luggage request list, thanks to its EventLuggageManagingSystem, and starts its path to go and collect the first piece of luggage. All this can be seen in fig.8.11.

At this time, both the Robots are following their paths, and both of them are keeping trace of the remaining pieces of luggage to manage. Robot002 is the first to reach its destination (see fig.8.12). Notice that the whole path is followed step by step through lines on the console. Also, it is possible to see when the Robot reach the first destination to collect luggage. A method that takes a few seconds simulates the physical action of collecting or depositing luggage.

In this case, Robot002 is the first to finish its task. So, after depositing its

```
Enviroment: Test2: Luggage to bring from (1,5) to (10,5) at 2015-01-23 16:05:57.313
Execution 7 of Robot001 starts at 2015-01-23 16:05:58.321
Robot001 updated active robot list:             [Robot2 is active]
Robot001 help request list:                     []
Robot001 luggage request list:                  []
Robot001 battery level: 100.0
MasterEnviromentManagingSystem: new event received and managed at 2015-01-23 16:05:58.794
Enviroment: Test3: Luggage to bring from (10,8) to (1,8) at 2015-01-23 16:05:58.321
Execution 7 of Robot002 starts at 2015-01-23 16:05:58.839
Robot002 updated active robot list:             [Robot1 is active]
Robot002 help request list:                     []
Robot002 luggage request list:                  [Luggage to bring from (1,5) to (10,5)]
Robot002 says: My position now is: (0,2) and my next position will be: (1,2)
Robot002 battery level: 100.0
MasterEnviromentManagingSystem: new event received and managed at 2015-01-23 16:05:59.306
Execution 8 of Robot001 starts at 2015-01-23 16:05:59.318
Robot001 updated active robot list:             [Robot2 is active]
Robot001 updated other robot position list:     [Position of Robot2 will be: (1,2)]
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (1,5) to (10,5),
                                                 Luggage to bring from (10,8) to (1,8)]
Robot1 is bringing luggage from (1,5) to (10,5)
Robot001 battery level: 100.0
Enviroment: Test4: Luggage to bring from (3,1) to (3,10) at 2015-01-23 16:05:58.824
MasterEnviromentManagingSystem: new event received and managed at 2015-01-23 16:05:59.793
Execution 8 of Robot002 starts at 2015-01-23 16:05:59.83
Robot002 updated active robot list:             [Robot1 is active]
Robot002 help request list:                     []
Robot002 luggage request list:                  [Luggage to bring from (10,8) to (1,8),
                                                 Luggage to bring from (3,1) to (3,10)]
Robot002 says: My position now is: (1,2) and my next position will be: (2,2)
Robot002 battery level: 100.0
```

Figure 8.11: Arrival of two other pieces of luggage.

```
Execution 24 of Robot001 starts at 2015-01-23 16:06:15.317
Robot001 updated active robot list:             [Robot2 is active]
Robot001 updated other robot position list:     [Position of Robot2 will be: (3,10)]
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (10,8) to (1,8),
                                                 Luggage to bring from (3,1) to (3,10)]
Robot001 says: My position now is: (9,5) and my next position will be: (10,5)
Robot001 battery level: 72.0
Execution 24 of Robot002 starts at 2015-01-23 16:06:15.83
Robot002 updated active robot list:             [Robot1 is active]
Robot002 updated other robot position list:     [Position of Robot1 will be: (10,5)]
Robot002 help request list:                     []
Robot002 luggage request list:                  [Luggage to bring from (10,8) to (1,8),
                                                 Luggage to bring from (3,1) to (3,10)]
Robot002 says: Ready to deposit luggage in position (3,10)
Robot002 says: Luggage deposited.
```

Figure 8.12: Robots following their paths.

first piece of luggage, it goes for the second one (see fig.8.13). Notice that also the luggage list of Robot001 is updated.

```
Robot2 is bringing luggage from (10,8) to (1,8)
Robot002 battery level: 66.0
Execution 26 of Robot001 starts at 2015-01-23 16:06:17.317
Robot001 updated active robot list:             [Robot2 is active]
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (3,1) to (3,10)]
```

Figure 8.13: Robots deposit and collect new luggage.

After a while, Robot002 has low battery and, as it is not busy, starts its path to charge position on its own. In the meanwhile, Robot001 is finishing its job bringing the second piece of luggage to destination (see fig.8.14).

```
Execution 48 of Robot002 starts at 2015-01-23 16:06:39.83
Robot002 updated active robot list:             [Robot1 is active]
Robot002 updated other robot position list:     [Position of Robot1 will be: (3,9)]
Robot002 help request list:                     []
Robot002 luggage request list:                  []
Robot002 has low battery and is going under charge...
Robot002 says: My position now is: (1,8) and my next position will be: (1,7)
Robot002 battery level: 20.0
```

Figure 8.14: Robot002 starts its path to charge position.

At the end of its path, Robot002 reaches charge position. Robot001, which has also battery low, starts its path to charge position on its own (see fig.8.15).

Then, both the Robots will call for a substitute, as seen in the previous test. After a while, Robots will be fully charged. In fig.8.16 it is possible to see how Robots keep their execution while charging. Once a Robot has fully charged, it will be ready again, help also old robots which sent the request and eventually get ready for bringing new luggage. First, Robot002 awakes and gets ready to help itself, then, it helps also Robot001, trying to awake another robot to be the substitute of Robot001.

This process is repeated continuously. No bugs and errors to signal during this

```
Robot002 says: I've arrived to charge position.
Execution 56 of Robot001 starts at 2015-01-23 16:06:47.317
Robot001 updated active robot list:             [Robot2 is active]
Robot001 help request list:                     [New help request from Robot2]
Robot001 luggage request list:                  []
Robot001 has low battery and is going under charge...
Robot001 says: My position now is: (3,6) and my next position will be: (3,5)
Robot001 needs to reach charge position as soon as possible!
Execution 56 of Robot002 starts at 2015-01-23 16:06:47.829
Robot002 updated active robot list:             [Robot1 is active]
Robot002 updated other robot position list:     [Position of Robot1 will be: (3,5)]
Robot002 help request list:                     [New help request from Robot2]
Robot002 luggage request list:                  []
Robot002 battery level: 16.0
Execution 57 of Robot001 starts at 2015-01-23 16:06:48.317
Robot001 updated active robot list:             [Robot2 is active]
Robot001 help request list:                     [New help request from Robot2]
Robot001 luggage request list:                  []
Robot001 has low battery and is going under charge...
```

Figure 8.15: Robot002 reaches charge position.

```
Execution 65 of Robot002 starts at 2015-01-23 16:06:56.829
Robot002 updated active robot list:             [Robot1 is active]
Robot002 help request list:                     [New help request from Robot2,
                                                 New help request from Robot1]
Robot002 luggage request list:                  []
Robot002 is now fully charged and ready (but still asleep)
Robot2 is a new healer for Robot2
Robot002 battery level: 100.0
Execution 66 of Robot001 starts at 2015-01-23 16:06:57.316
Robot001 updated active robot list:             [Robot2 is active]
Robot001 help request list:                     [New help request from Robot1]
Robot001 luggage request list:                  []
Robot001 battery level: 20.0
Execution 66 of Robot002 starts at 2015-01-23 16:06:57.829
Robot002 updated active robot list:             [Robot1 is active]
Robot002 help request list:                     [New help request from Robot1]
Robot002 luggage request list:                  []
Robot002 is now fully charged and ready (but still asleep)
Robot002 is a healer and now is awake
Robot002 battery level: 100.0
Execution 67 of Robot001 starts at 2015-01-23 16:06:58.316
Robot001 updated active robot list:             [Robot2 is active]
Robot001 help request list:                     [New help request from Robot1]
Robot001 luggage request list:                  []
Robot001 battery level: 30.0
Execution 67 of Robot002 starts at 2015-01-23 16:06:58.828
Robot002 updated active robot list:             [Robot1 is active]
Robot002 help request list:                     [New help request from Robot1]
Robot002 luggage request list:                  []
Robot2 is a new healer for Robot1
Robot002 battery level: 100.0
Execution 68 of Robot001 starts at 2015-01-23 16:06:59.318
Robot001 updated active robot list:             [Robot2 is active]
Robot001 help request list:                     []
Robot001 luggage request list:                  []
Robot001 battery level: 40.0
Execution 68 of Robot002 starts at 2015-01-23 16:06:59.828
Robot002 updated active robot list:             [Robot1 is active]
Robot002 help request list:                     []
Robot002 luggage request list:                  []
Robot002 is a healer and awakes another robot
```

Figure 8.16: Robot002 is fully charged and ready again.

test. Also, a collision has been avoided (see section 8.5 for details).

The complete robots' path can be seen in fig.8.17.

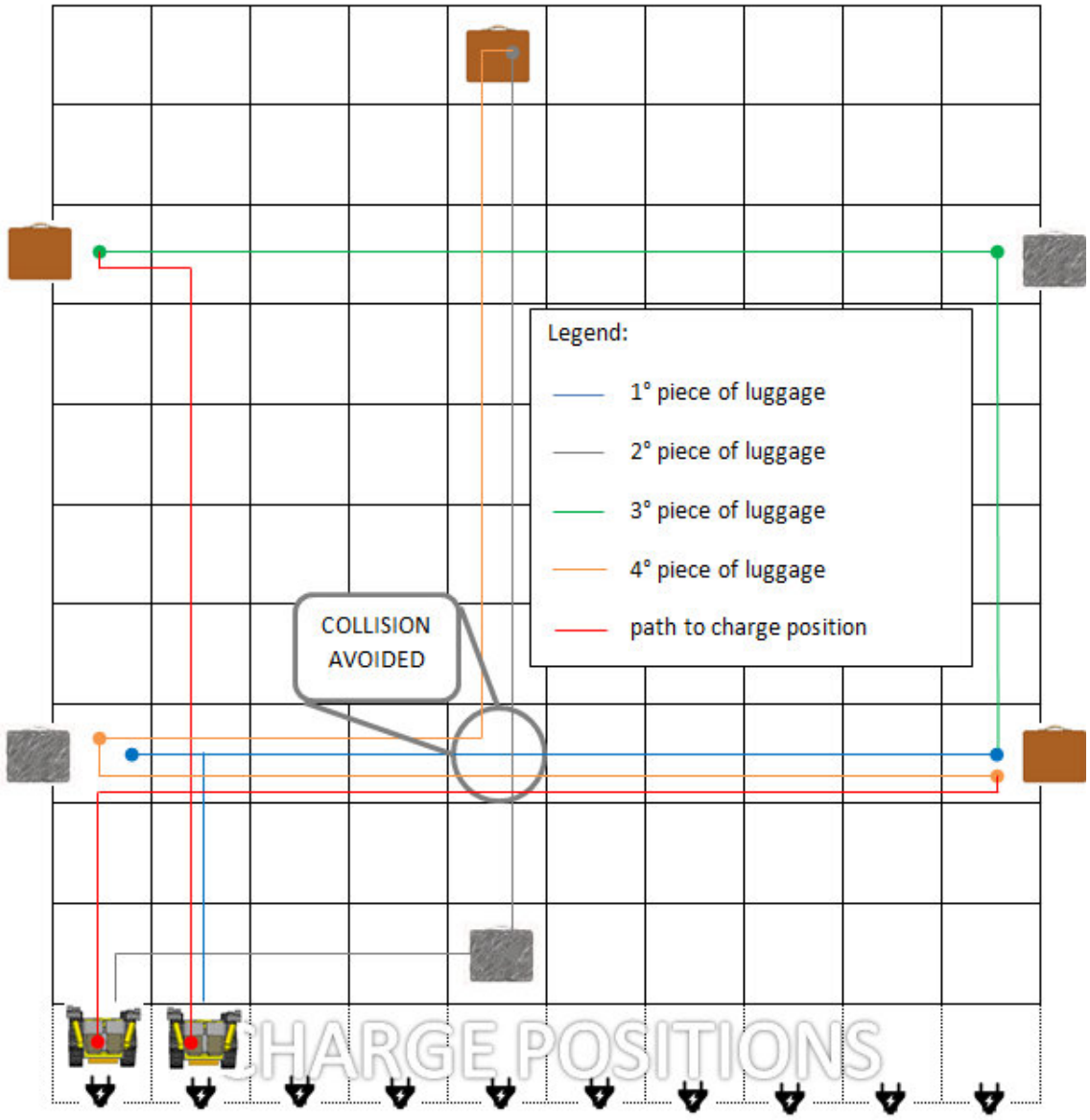

Figure 8.17: Robots' complete paths.

8.3 Testing two Robots with several pieces of luggage after an airplane landing

Src code enclosed under the name of *SAOMS_test03*.

As already explained, the components are instantiated. Then, one execution for each period of each component occurs. This fact is highlighted on the console through a line with a time stamp indication. Please, notice that each execution starts with the precision of a hundredth of a second.

Please see again section 8.2 for further details.

Then, an exceptional event is revealed and managed: a landing (see fig.8.18).

```
Enviroment: Test1: New event: LANDING with 4 expected pieces of luggage.
Execution 3 of Robot001 starts at 2015-01-23 16:59:20.841
Robot001 updated active robot list:             [Robot2 is active]
Robot001 help request list:                     []
Robot001 luggage request list:                  []
Robot001 battery level: 100.0
MasterEnviromentManagingSystem: new event received and managed at 2015-01-23 16:59:21.315
Execution 3 of Robot002 starts at 2015-01-23 16:59:21.365
Robot002 updated active robot list:             [Robot1 is active]
Robot002 help request list:                     []
Robot002 luggage request list:                  []
Robot002 battery level: 100.0
MasterRobotManagingSystem updated active robot list:            [Robot1 is active,
                                                                 Robot2 is active]
```

Figure 8.18: Airplane landing.

The MasterRobotManagingSystem is the one which intervene and manage the awakening of new robots due to the number of expected pieces of luggage.

Except of the testing of this feature, the rest of the test is similar to the previous one. No bugs and errors to signal during this test.

8.4 Stress Test: 100 pieces of luggage and 10 simultaneous cooperating robots

The last test has been done in a stressful situation: 100 pieces of luggage have been simulated in the system and up to ten cooperating robots have been provided. No errors detected, many collisions avoided.

Please check testing code attached to this documentation for further details and examples.

8.5 Collision avoidance

During testing, sometimes, robots' paths intersect each other's ones. When it happens at the same time there is a serious risk of collision. Thanks to CollisionManagingSystem, robots avoid crashes, following the master/slave policy with ID priority (the lower ID, the higher priority). In fig.8.19 it is possible to see the perfect timing and functionality of this ability. Robot002 communicates (as every execution) its next position. Robot001 receives this message and send a message to say that its next position will be (accidentally) the same of Robot002. Even if Robot002 sent the message with the intention of occupy block (3,5) first, the minor ID priority is applied: this to avoid crashes caused by (unlikely) lateness of JMS. In fact, it is possible to notice that Robot002 does not move during execution 13: Robot001 knows that Robot002's next position will be block (3,5) during execution 14 and also during execution 15.

8.6 Free testing

Other tests have been done with no other salient aspect to highlight, due to robustness of the project.

8.7 Considerations after testing

What came up testing the software concerns the necessity of a high computational power depending on the MessageBroker to manage messages and the whole JMS system. In fact, after a few tests on a personal computer, a reboot is needed.

```
Execution 17 of Robot001 starts at 2015-01-23 16:06:08.317
Robot001 updated active robot list:             [Robot2 is active]
Robot001 updated other robot position list:     [Position of Robot2 will be: (3,5)]
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (10,8) to (1,8)]
Robot001 says: My position now is: (2,5) and my next position will be: (3,5)
Robot001 battery level: 86.0
Execution 17 of Robot002 starts at 2015-01-23 16:06:08.83
Robot002 updated active robot list:             [Robot1 is active]
Robot002 updated other robot position list:     [Position of Robot1 will be: (3,5)]
Robot002 help request list:                     []
Robot002 luggage request list:                  [Luggage to bring from (10,8) to (1,8)]
Robot002: RISK OF COLLISION DETECTED AND AVOIDED!
Robot002 battery level: 82.0
Execution 18 of Robot001 starts at 2015-01-23 16:06:09.316
Robot001 updated active robot list:             [Robot2 is active]
Robot001 updated other robot position list:     [Position of Robot2 will be: (3,5)]
Robot001 help request list:                     []
Robot001 luggage request list:                  [Luggage to bring from (10,8) to (1,8)]
Robot001: RISK OF COLLISION DETECTED AND AVOIDED!
Robot001 says: My position now is: (3,5) and my next position will be: (4,5)
Robot001 battery level: 84.0
Execution 18 of Robot002 starts at 2015-01-23 16:06:09.829
Robot002 updated active robot list:             [Robot1 is active]
Robot002 updated other robot position list:     [Position of Robot1 will be: (4,5)]
Robot002 help request list:                     []
Robot002 luggage request list:                  [Luggage to bring from (10,8) to (1,8)]
Robot002 says: My position now is: (3,5) and my next position will be: (3,6)
Robot002 battery level: 80.0
```

Figure 8.19: Collision avoidance.

Chapter 9

Final Considerations

This project has been created as a basic tool to be extended for specific purposes. In fact, it is simply adjustable to satisfy specific intention such as, for example, adding new methods or override already existing ones. This would seamlessly integrate new abilities with the already existing environment.

Please note that there is the possibility for the user to manage every single aspect of the project simply sending messages on correct topics, also at runtime.

9.1 Robustness

The robustness of the software relies on the contemplation of all the possible scenarios in which tasks and Robots can find themselves. Particular attention is dedicated to the possibility for each Robot to call another Robot to ask for help, with no need to ask the MasterSystem. This is provided by CriticalSituationManagingSystem MAPE-K class. Also, more critical situations as fatal breakages are managed in local by robots themselves, not compromising the overall functionality of the software.

9.2 Flexibility

The flexibility of this project depends on the possibility of choosing the number of Robots depending on the workload. For example, when a new airplane lands, the MasterRobotManagingSystem awakes as many Robot as needed to manage the new pieces of luggage. This is done by the AwakeAsleepManagingSystem MAPE-K class. On the contrary, Robot themselves go under charge once the busy situation is managed and the workload becomes normal again.

Once again, the system can be managed simply by sending messages on correct topics also at runtime.

9.3 Safety

The safety of this project is guaranteed by the CollisionManagingSystem MAPE-K class. It predicts collisions (also between more than one robot) and avoid them.

9.4 Maintainability

Thanks to its deep communication unit, (hardware and software) failures are promptly forwarded to the proper authorities. This allows a self-adaptive maintainability, as far as a robot component is able to move itself to charge position. Anyway, if the Robot is not able to reach charge position, safety systems are activated and the maintenance technician is advise in good time.

SAOMS project has been developed with the lowest expectations about hardwares: a system with no sensors or cameras has been considered. Please see section 9.7 for further extension's ideas.

What came up with testing is the need of a good communication unit, in order

to sustain the messaging service computational workload.

9.5 Real-Time System

As this project is based on a JMS communication and mutual messages are constantly and concurrently exchanged within a certain time period, it can be regarded as a real-time system. In fact, except for physiological negligible messaging delays, each robot accomplishes its tasks before the end of its period. Anyway, these delays do not become a problem as the period is not affected by them: thanks to its schedulation, a component executes periodically with tiny margin of error.

The margin or error, due to the Java platform, can be measured in term of milliseconds. In fact, deep in Task implementation (in TCA library), there is a line defining this margin: *future=scheduler.scheduleAtFixedRate(component, delay, period, TimeUnit.MILLISECONDS);*. What just said is reflected in practical measurements, that can be seen in a screenshot of a real execution in fig.9.1.

9.6 Applicability in reality

This project has been thought as an academic software to develop and test a reproduced scenario. However, albeit with some limitation due to the physical time need by the robots to move, this application may be tested in a real environment. In fact, supposing an appropriate execution period for each component, conflicts are still managed as well as paths to follow, luggage to collect, etc. Anyway, guarantees are the same ones as JMS and the java.util.concurrent library provide. Draw your own conclusion.

```
Execution of Robot001 at 2014-10-02 10:25:11.955
Execution of Robot001 at 2014-10-02 10:25:12.955
Execution of Robot001 at 2014-10-02 10:25:13.955
Execution of Robot001 at 2014-10-02 10:25:14.955
Execution of Robot001 at 2014-10-02 10:25:15.956
Execution of Robot001 at 2014-10-02 10:25:16.954
Execution of Robot001 at 2014-10-02 10:25:17.956
Execution of Robot001 at 2014-10-02 10:25:18.955
Execution of Robot001 at 2014-10-02 10:25:19.954
Execution of Robot001 at 2014-10-02 10:25:20.955
Execution of Robot001 at 2014-10-02 10:25:21.955
Execution of Robot001 at 2014-10-02 10:25:22.954
Execution of Robot001 at 2014-10-02 10:25:23.955
Execution of Robot001 at 2014-10-02 10:25:24.955
Execution of Robot001 at 2014-10-02 10:25:25.954
Execution of Robot001 at 2014-10-02 10:25:26.955
Execution of Robot001 at 2014-10-02 10:25:27.956
Execution of Robot001 at 2014-10-02 10:25:28.954
Execution of Robot001 at 2014-10-02 10:25:29.955
Execution of Robot001 at 2014-10-02 10:25:30.955
Execution of Robot001 at 2014-10-02 10:25:31.955
Execution of Robot001 at 2014-10-02 10:25:32.955
Execution of Robot001 at 2014-10-02 10:25:33.956
Execution of Robot001 at 2014-10-02 10:25:34.956
Execution of Robot001 at 2014-10-02 10:25:35.954
Execution of Robot001 at 2014-10-02 10:25:36.955
```

Figure 9.1: Measurements of time precision of execution.

9.7 Future Directions

At the state of the art, no systems are designed to recovery robots extremely critical situations such as the ones with terminal hardware breakages. SAOMS, in this unfortunate event, will continue working properly as new robots are taking the damaged one place. An idea could be the one of creating a subsystem for healing robots with the particular task of recovering damaged robots.

Moreover, as SAOMS has been developed on a minimal hardware system, new functionalities such as introducing sensors' captions to avoid crashes also in case of an external obstacle, or integrating a GPS system to reduce the margin of error during path following, are strongly recommended: nothing easier and faster than override execute() method of the interested robots. See next chapter to find out how.

Chapter 10

User Extensions

For each robot movement needed, a public method is provided to be overridden. Moreover, to add a periodic tastk it is simply enough to work on the execute() method, instead, for a task to be run once at the star of the robot, the method to work on is the initialize(). To make a component becoming a publisher on and/or a subscriber to selected topics it is necessary to work on the initialize method. Then, for repetitive tasks, it is suggested to override the execute method. For example, if a robot has a camera and wants to publish a snap-shot on a *CameraTopic* once each period, listing 10.1 could be a perfect example of how to implement this functionality.

```
1  @Scope("COMPOSITE")
2  public class Robot999 extends RobotImpl
3                                        implements Serializable{
4      public Robot001(){
5      super();
6      BlockPos cp = new BlockPos(0, 1); // charge position
7      this.robID  = new RobotID ("Robot999", 999);
8      this.setBatDischarge(2);
9    }
10
11   @Override
12   public void initialize{
13     super.initialize();
14     publish(cameraTopic);
```

```
15    }
16
17    @Override
18    public void execute{
19      super.execute();
20      // get a snap-shot
21      send(cameraTopic, new Message(snap-shot));
22    }
23  }
```

Listing 10.1: Example of MasterEnvironmentMS extension.

Previously an example of how to manage new not yet categorized events has been shown (see listing 6.13).

MAPE-K classes can be extended as well as each dedicated MAPE-K onboard class extends the abstract MAPEKclass, as shown in listing 10.2.

```
1   MAPE-K class UserMS{
2     // Additional Fields
3     Object object;
4
5     @Override
6     public void callMAPE(Object obj){
7       super.callMAPE();
8       this.object=obj;
9     }
10    @Override
11    public void monitor() {
12      //..do some user computation to monitor system
13    }
14    @Override
15    public void analyze() {
16      //..do some user computation to analyze system
17    }
18    @Override
19    public void plan() {
20      //..do some user computation to plan actions
21    }
22    @Override
```

```
23    public void execute() {
24      //..do some user computation to execute actions
25    }
26  }
```

Listing 10.2: Example of MasterEnvironmentMS extension.

In the end, for each new robot component, the user will have to create a new dedicated xml .composite file, as explained in section 6.6.1.

Chapter 11

Curiosities

The final working version of SAOMS is version 030.

The final number of lines is major than 3000 and depends on different tests.

The project has been conceived in 2013 and developed during year 2014. The final version has been released in early 2015.

A reduced number of people have been working on it: Yamuna Maccarana and Umberto Paramento, under the supervision of prof.Patrizia Scandurra.

Bibliography

[1] GitHub online repository

Find src on github.com/yamunamaccarana

[2] D.Brugali; A.Fernandes da Fonseca; A.Luzzana; Y.Maccarana:

Developing Service Oriented Robot Control System

Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium

Get this paper on IEEE website

[3] D.Brugali; A.Fernandes da Fonseca; A.Luzzana; Y.Maccarana:

The Task Component Architecture

See src on github.com/yamunamaccarana/TCA

[4] *Extensible Markup Language (XML) 1.0 (Fifth Edition)*

W3C Recommendation 26 November 2008

[5] S.Laws; M.Combellack; R.Feng; H.Mahbod; S.Nash:

Tuscany SCA in Action

Manning

[6] *Tuscany distribution*

See tuscany.apache.org/getting-started-with-tuscany.html

[7] D.K.Barry with D.Dick:

Web Services, Service-Oriented Architectures, and Cloud Computing: The Savvy Manager's Guide (Second Edition)

MK

[8] *The Java Language Environment*

1.2 Design Goals of the *Java*^{TM} Programming Language. Oracle. 1999-01-01. Retrieved 2013-01-14.

[9] *Open Robot Control Software Project*

See orocos.org