# Multi-Agent Approach to Modeling and Implementing Fault-Tolerance in Reactive Autonomic Systems

Nassir Shafiei-Dizaji

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science (Software Engineering) at

Concordia University

Montreal, Quebec, Canada

April 2011

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Nassir Shafiei-Dizaji

Entitled: Multi-Agent Approach to Modeling and Implementing Fault-Tolerance in Reactive Autonomic Systems

and submitted in partial fulfillment of the requirements for the degree of

**Master of Applied Science (Software Engineering)**

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Yuhong Yan

_____ Examiner
Dr. Peter Grogono

_____ Examiner
Dr. Joey Paquet

_____ Supervisor
Dr. Olga Ormandjieva

_____ Supervisor
Dr. Jamal Bentahar

Approved by _____
Chair of Department or Graduate Program Director

_____
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Date _____

# Abstract


## Multi-Agent Approach to Modeling and Implementing Fault-Tolerance in Reactive Autonomic Systems

Nassir Shafiei-Dizaji

Recently, autonomic computing has been proposed as a promising solution for software complexity in IT industry. As an autonomic approach, the Reactive Autonomic Systems Framework (RASF) proposes a formal modeling based on mathematical category theory, which addresses the self-* properties of reactive autonomic systems in a more abstract level.

This thesis is about the specification and implementation of the reactive autonomic systems (RAS) through multi-agent approach by laying emphasis on the fault-tolerance property of RAS. Furthermore, this thesis proposes a model-driven approach to transform the RAS model to agent templates in multi-agent model using Extensible Stylesheet Language Transformation (XSLT). The multi-agent approach in this research is implemented by Jadex, a high-level Java-based agent programming language. The intelligent agents are created in Jadex based on the Belief-Desire-Intension (BDI) agent architecture. The approach is illustrated on a case study.

# Acknowledgments

This thesis would not have been possible without the guidance and help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this study.

First and foremost, my utmost gratitude to my supervisors, Dr. Olga Ormandjieva and Dr. Jamal Bentahar whose encouragement, supervision, and support from the preliminary to the concluding level enabled me to hurdle all the obstacles in the completion of this research. Also, I am heartily thankful to all members of our research group, especially Dr. Stan Klasa and Heng Kuang for their comments and support.

In addition, I would like to express my appreciation to my examiners, Dr. Peter Grogono and Dr. Joey Paquet, for their precious time to review my thesis and give me helpful advice.

I also would like to thank Concordia University and the Faculty of Computer Science and Software Engineering for offering me the precious opportunity and excellent academic environment to achieve this work.

Finally, I would like to dedicate my ultimate thanks to my beloved wife Afsoon and my son Arian for their encouragement and never ending support.

# Table of Contents

# List of Figures

# List of Abbreviations

RAS ..... Reactive Autonomic System

RASF ..... Reactive Autonomic System Framework

MAS ..... Multi-Agent System

RAE ..... Reactive Autonomic Element

RAO ..... Reactive Autonomic Object

RAOL ..... Reactive Autonomic Object Leader

RAC ..... Reactive Autonomic Component

RACG ..... Reactive Autonomic Component Group

RACS ..... Reactive Autonomic Component Supervisor

RACGM ..... Reactive Autonomic Component Group Manager

ICL ..... Intelligent Control Loop

CT ..... Category Theory

BDI ..... Belief-Desire-Intention

LHS ..... Left Hand Side

RHS ..... Right Hand Side

ADF ..... Agent Definition File

MT ..... Model Transformation

XML ..... Extensible Markup Language

XSLT ..... Extensible Stylesheet Language Transformation

GS ..... Group Supervisor

GM ..... Group Manager

# Chapter 1: Introduction

This thesis is about implementing the fault-tolerance property of Reactive Autonomic Systems (RAS) using Multi-Agent Systems (MAS) by Jadex, which is a high-level agent-programming tool as well as transforming the RAS abstract meta-model to an implementable MAS meta-model using Extensible Stylesheet Language Transformation (XSLT).

In this chapter, we will discuss the research questions, proposed approach, and motivation behind the selection of MAS to implement RAS using Jadex as a programming environment to develop intelligent agents in MAS. Moreover, our contribution in defining and modeling fault-tolerance property of RAS as MAS and implementing it with Jadex will be illustrated. Furthermore, the idea of developing the rules to transform the RAS meta-model to a MAS meta-model using XML model transformation framework will be presented. These transformation rules are used to produce Jadex agent templates with fault-tolerance property in MAS. Finally, we will present the outline of this thesis.

## 1.1. Context of Research

**Autonomic Computing.** Autonomic computing is considered as one solution for today's software problems such as excessive software complexity and enormous maintenance load [14, 22]. The primary goal of autonomic computing is self-management, which can be further decomposed into self-configuration, self-healing, self-optimization and self-protection [14]. The absence of a formal framework for autonomic systems based on a strong theoretical backbone has encouraged the authors of [2] to propose Reactive Autonomic Systems Framework (RASF) specified using the mathematical Category Theory (CT). The CT is an abstract theory that examines mathematical concepts and their relationships by formalizing them as *objects* and *arrows* (*morphisms*) [23]. The different definitions and axioms of CT have been used to specify architectural and behavioral aspects of RASF. For example, the Reactive Autonomic Elements (RAE), which are the atomic components of RASF, are mapped to *objects* and their interactions to *morphisms* in CT.

Since the RAS framework is very abstract, it needs to be implemented using the available current approaches for autonomic paradigm. One of the proper solutions to model and implement the autonomic systems seems to be the multi-agent approach since the autonomous behavior of intelligent agents can be easily mapped to self-* properties in RAS [25]. On the other side, plenty of frameworks and tools are available in the MAS domain to be acquired and utilized.

**Multi-Agent Systems.** A Multi-Agent System (MAS) [9] is a software system consisting of a group of intelligent agents capable of autonomous actions that interact with each other through a given Agent Communication Language (ACL) [11] to achieve a specific

goal. The agents in the MAS cooperate, coordinate or negotiate with each other to achieve the goals that are difficult for a single agent to accomplish. The reactive and proactive properties of the intelligent agents as well as their cooperative or self-interested behaviors make them the ideal mapping for RAE in RAS framework. The Belief-Desire-Intention (BDI) architecture is one of the well-known existing architectural models for MAS [9]. In BDI architecture, the rational agents [15] are defined as a philosophical model having specific notations for: 1) Beliefs: indicate the knowledge of the agent about its environment and other agents; 2) Desires: specify the goals that the agent may achieve; and 3) Intentions: indicate what the agent has chosen to accomplish as a plan [16]. In BDI model, the plans are triggered by internal goals as well as external messages from the environment or other agents. As a well-defined architecture for multi-agent systems, BDI has become the basis for a number of agent-oriented programming tools such as Jadex [26].

**Jadex.** Jadex is a Java-based agent programming middle-ware that is based on Java Agent Development Framework (JADE) and complies with Foundation for Intelligent Physical Agents (FIPA) standard for agent communication. The architecture of Jadex follows the BDI model to define beliefs, goals, plans and message events for the intelligent agents. Each agent in Jadex is specified in an XML file called Agent Definition File (ADF) containing corresponding tags for beliefs, goals, events and plan headers. The body of the corresponding plans for each agent is defined in separate Java class files. These plans are triggered by internal goals specified in the ADF or by external message events coming from the environment or other agents [13]. The execution of the plans may modify beliefs, create new goals, send external messages or run other plans.

Also, it is possible to define common characteristics of agents in files called capability ADF having corresponding plan class files. These capabilities can be included in agents and used as needed. The standard and tangible architecture of Jadex permits developers to easily define agents and model complicated agent structures. The Java-based language of Jadex allows using it with any Java development environment such as Eclipse and taking advantage of all Java features. On the other hand, the XML format of the ADF allows for easy development and use of this file as a convenient output model for XML-based model transformation framework to convert the RAS meta-model to the MAS meta-model.

**Model Transformation.** Model Transformation (MT) is the process of converting one specific model to another model. In this process, the input model is called Left Hand Side (LHS) that conforms to a meta-model and the output model is called Right Hand Side (RHS) that conforms to another meta-model. To automate this process, there are some model transformation tools such as Extensible Stylesheet Language Transformation (XSLT) [45]. XSLT is an XML tool to define the transformation mappings from one input XML file to another XML or Text file. XSLT uses XPath language to define the rules and algorithms that are used in the conversion operation.

In this thesis, we will define a grammar to create the XML representation of RAS, which will serve as an input model for XSLT. Both the architectural and behavioral aspects of RAS with regard to fault-tolerance to achieve self-healing property of RAE are serialized in XML format. The developed transformation rules will take RAS as input and convert it to the corresponding templates of agents and plans in Jadex.

## 1.2. Motivations

The Reactive Autonomic Systems (RAS) framework uses very rich bases from mathematical Category Theory (CT) to define its static architectural structure and dynamic behavioral model. Each of the elements in the RAS framework is defined in a high abstraction level with categorical *objects* and the interaction between them is mapped to *morphisms* between these objects in corresponding categories [19]. The work done until now focuses on proving different autonomic properties in the RAS framework using mathematical axioms and theorems in category theory. For instance, in a very abstract level of fulfilling the fault-tolerance property, by using categorical properties, it is asserted that a Reactive Autonomic Element (RAE) can be replaced with a similar RAE (the substitutability property) [19]. This level of abstraction in terms of the necessity to implement and test what is proven mathematically requires a framework to put into practice the RAS framework.

Our first motivation to use MAS as a mapping for RAS is to implement and test the properties of the RAS framework. It is obvious that the considerations to select MAS are based on the inherent similarities between the two frameworks and the enormous volume of work done in multi-agent community. Another reason that has encouraged us to choose MAS is the number of powerful tools to implement intelligent agents. It was very important to take advantage of a tool that is easy to use for modeling complex concepts. Jadex, a rich Java-based agent programming language, is an appropriate tool to map RAS models to MAS models. Some concepts in Jadex have helped us review the representation model of RAS in XML format and bring some changes in some cases.

Our second motivation is to propose a Model Transformation (MT) framework to produce multi-agent templates representing the RAS components that satisfy the self-healing property. RAS presently has the form of architectural and sequence diagrams. The proposed framework defines a grammar for all components in RAS and serializes them into XML format. The result serves as an input to the MT framework that transforms it into agent template representations in Jadex considering in particular the fault-tolerance property of RAS. These templates can be used as blue-prints to design different systems that intend to comply with the RAS framework.

## 1.3. Research Questions

The research questions we are aiming to address in this work are listed next:

1. How can RAS specifications of self-* properties be refined into MAS models? How do we refine the specifications of the RAS meta-model such as RAE using the MAS concepts such as intelligent agents?
2. What is the best multi-agent architecture to choose for this purpose? What programming tool can we select conforming to this architecture?
3. Can model transformation approach be applied to transform a model conforming to a RAS meta-model into a MAS meta-model? What is the best model transformation framework to be applied?

## 1.4. Proposed Approach and Contributions

The main goal of this thesis is the implementation of RAS with MAS using a model transformation framework to create necessary agent templates. In fact the contributions of this work are as follows:

1. Mapping the RAS structures to the corresponding MAS components [Chapter 3]: The atomic element in the RAS meta-model is Reactive Autonomic Object (RAO) which is mapped to an agent in the MAS meta-model. This means that for each RAO in the RAS meta-model, we create an ADF file in XML format containing the appropriate tags for beliefs, goals, plan headers and event messages. The composite elements of the RAS meta-model such as the Reactive Autonomic Component (RAC) are in fact the combination of RAO elements with proper communications between them. The mapping for these more elaborated structures in MAS is a series of corresponding agents that are capable to communicate with each other respectively.
2. Mapping the behavioral model of RAS to MAS [Chapter 3]: The behavior of the RAS meta-model is captured using sequence diagrams. Using the Marsworld case study [5], we implement the fault-tolerance property of the RAS meta-model that involves the substitutability property of elements in the RAS framework to assert what was proven mathematically in [19]. Consequently, the reactive and proactive comportments of RAS elements for the fault-tolerance property are merged with the plan files.
3. Using Jadex to implement the MAS model of RAS [Chapter 4]: Among all agent architectures, the Beliefs-Desires-Intentions (BDI) model is the appropriate choice for our approach. The justification is given in Chapter 2. In this model, we specify beliefs as the knowledge of the agent, desires as the goals to be fulfilled and the intentions as the plans to achieve the goals. An appropriate agent programming tool that is based upon BDI architecture is Jadex, which is a Java-

based solution. Using Jadex, we have implemented Marsworld case study to prove the fault-tolerance property.

4. Proposing a model transformation framework to develop agent templates [Chapter 3]: Since Jadex uses XML format to define its agent profiles, it can serve as a suitable output format for model transformation frameworks such as Extensible Stylesheet Language Transformation (XSLT) that is based on XML input and output. For this purpose, we have defined a grammar to capture the RAS concepts in XML format that provide the input for our model transformation framework. This framework defines transformation rules that take the RAS meta-model concepts in XML format and transforms them to the BDI-based MAS agents in Jadex.

Figure 1.1 illustrates the proposed approach and the contribution of this thesis.

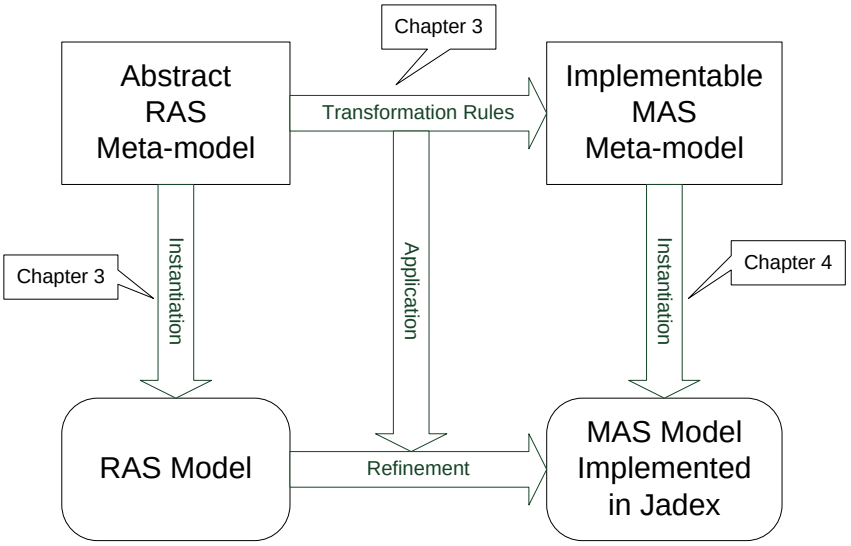


**Figure 1.1:** The schema of the proposed approach.

## 1.5. Outline

This thesis contains 6 chapters. Chapter 2 introduces the background that includes the description of the RAS framework, category theory, MAS, the Marsworld case study involving fault-tolerance property, the Jadex programming tool and the XSLT model transformation framework. Chapter 3 provides the main contribution describing the transformation process from RAS to MAS including the RAS grammar, input model, transformation rules, and output model. Chapter 4 illustrates the consideration of fault-tolerance property using the Marsworld case study. In Chapter 5, we present some related work in autonomic computing, multi-agent systems and model transformation areas. Finally, in Chapter 6, we summarize our main contribution and identify directions for future work.

# Chapter 2: Background

## 2.1. Introduction

Software systems are increasing in size and complexity so that their development and maintenance become more complicated. The present software solutions do not respond to this rapid change because the management of this mass of complexity goes beyond the capabilities of IT professionals. More and more the companies are looking for solutions that reduce human intervention in complex and time consuming tasks. The solution to this problem is a system that helps managers deal only with high-level critical tasks and handles low-level complex job itself. As a concrete example, we refer to the planet Mars exploration missions that are accomplished by robots. The message transmission between earth and Mars takes a long time and scientists need to minimize this communication and limit it to messages about crucial decisions. As a result, the robots must depend on their own intelligence. In fact, they have to be autonomous enough to carry out their own tasks. One solution is a system which manipulates a large number of inexpensive robots. These robots with simple capabilities are grouped together to form intelligent swarms. The coordination of actions between these robots is critical and a formal framework for their autonomic behavior is needed.

Many formal methods for swarm systems with autonomic behaviors are compared in [1]. However, according to the paper [19]: 1) it is not possible to come up with one single formal method that satisfies all necessary properties; 2) the proposed specifications cannot be easily transformed to program code; and 3) they cannot be applied as input to model checkers for automatic verification purposes. For proving the correctness of self-* properties, [2, 19] propose a formal framework named Reactive Autonomic Systems Framework (RASF) that tries to resolve the mentioned problems.

RASF is a formal framework for modeling reactive autonomic systems with self-* properties. The self-* properties consist of self-management that can be attained by realizing self-configuration, self-healing, self-optimization and self-protection [14]. RASF is based on the mathematical Category Theory (CT) that models Reactive Autonomic Systems (RAS) [2]. The authors of [19] use different applications such as Mars case study to build a RAS meta-model with regard to CT. The RAS meta-modeling provides properties and constraints that are correct by construction rules based on formal verification embedded in the backend CT models [2].

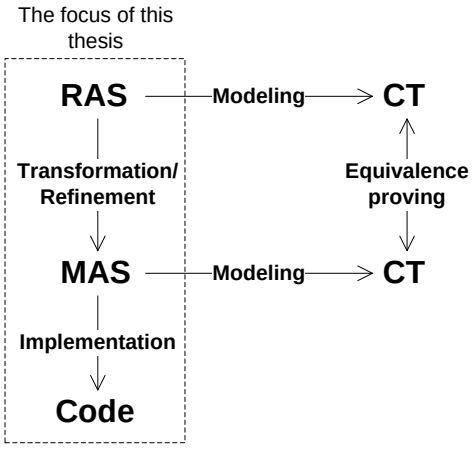


**Figure 2.1:** The schema of the whole project and the focus of this thesis.

In this research, we implement RAS through Multi-Agent Systems (MAS). Among different MAS frameworks we have selected the BDI model based on Beliefs, Desires, and Intentions that is a deliberative agent architecture. Java based agent development frameworks are widely used to implement complex systems and we are using Jadex that is based on XML and the Java programming language to elaborate agents in BDI multi-agent model. Figure 2.1 depicts the whole picture of the project and the focus of this thesis.

Figure 2.1 illustrates that the RAS components as well as the MAS components can be represented by the categorical concepts of CT. Using this representation, the RAS components can be mapped to MAS elements such as agents. The resulting MAS model is implemented by a multi-agent programming language such as Jadex. The work done in [2, 3, 4] focuses on self-monitoring property of RAS. This thesis concentrates on fault-tolerance property with regard to substitutability property in RAS using Marsworld [5] as a motivating case study.

The organization of the rest of this chapter is as follows: Section 2.2 describes the Mars exploration case study, which is used to illustrate our approach. Section 2.3 explains the RAS framework. Section 2.4 gives a brief description about category theory. Section 2.5 makes a bridge between the RAS framework and CT in terms of substitutability property. Section 2.6 explains multi-agent systems and Jadex. In Section 2.7 fault-tolerance mechanism based on the substitutability property will be discussed. Section 2.8 describes the model transformation framework and its XML-based tools. Finally, Section 2.9 states our conclusions.

## 2.2. Case Study

The Marsworld [5] case study is used in the rest of this paper to illustrate our approach. In this case study, a group of robots accomplish ore exploitation on the planet Mars. To achieve this goal, these robots must locate ore resources in the area, mine them, and transport produced ore to a base. This process is completed by three types of robots. There is a *sentry robot* whose responsibility is to analyze suspicious spots to evaluate if there is enough ore to be mined. This type of robot has wider sensor range to better verify candidate locations. When the sentry robot evaluates a mine to be exploited, it sends its location to a second robot type known as *production robot*. This robot has devices to dig and mine ore. After finishing its job, the production robot calls the *carry robot* to transport the produced ore to home base. The carry robot has the necessary equipments to carry ore and the ability to move faster than the other robots.

To better illustrate our approach, we have added two more types of robots to this case study. These two robots are more involved in administration and coordination tasks at the autonomic group level. The first robot of this type is *group supervisor robot*. The responsibility of this robot is to form exploitation groups, coordinate and validate its members. The second type of robots in a higher level of the hierarchy of robots is the *system manager robot*. This robot coordinates group supervisor robots, assigns mining tasks to them in different areas and can communicate with ground station on the Earth. These two robot types are very important since their jobs are critical to the system. They have their own backup robots and have access to the repositories of their own level.

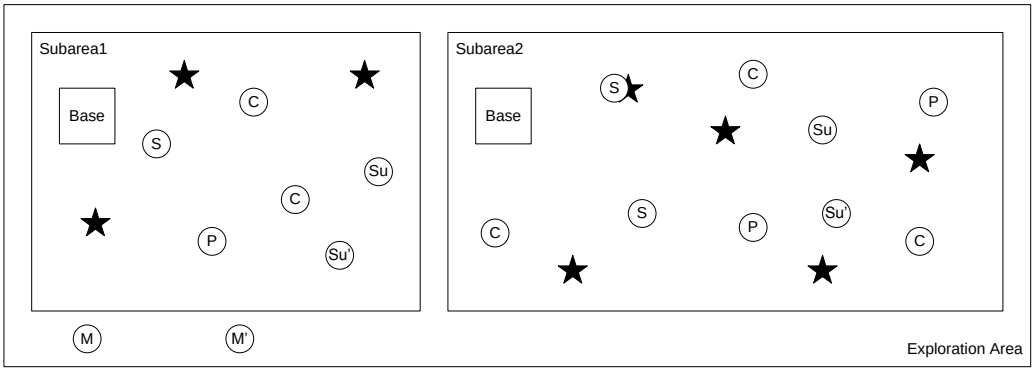


**Figure 2.2:** A sample scenario of the Marsworld case study.

Figure 2.2 shows a sample of the Marsworld scenario. In this example, the system manager robot after receiving corresponding commands from the Earth, assigns two mining areas to two group supervisor robots, *supervisor1* and *supervisor2*. According to some parameters of these two areas such as the surface and capacity of ore, the group supervisor robots form their own groups that consist of sentry robots, production robots and carry robots and start mining the areas. Also, the supervisor robots can form their groups by requesting more resources, such as any type of robots, from other group supervisor robots pending on their availability. During the mining process, the group supervisor robot checks instantly its group's members and also interchanges administrative messages with the system manager robot as well as other group supervisor robots.

## 2.3. Reactive Autonomic Systems Framework

In this section, we will explain the Reactive Autonomic Systems Framework (RASF) approach [2]. This is a formal approach for modeling reactive autonomic systems with

self-* properties that is based on the mathematical category theory. This section will demonstrate this framework from two perspectives: 1) architecture of the RAS meta-model; and 2) behavior of the RAS meta-model.

### 2.3.1. Architecture of the RAS meta-model

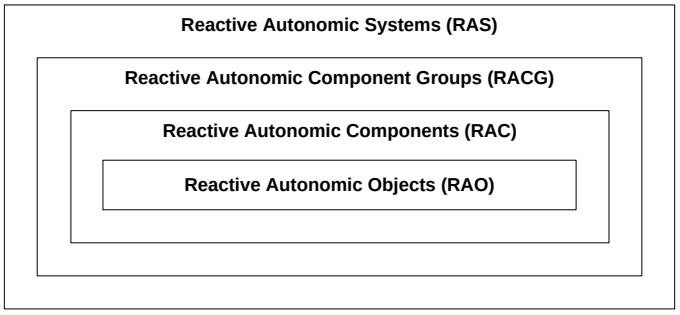


**Figure 2.3:** The architecture of the RAS meta-model.

The RAS architecture consists of four layers, the Reactive Autonomic Objects (RAO) being the simplest element, the Reactive Autonomic Components (RAC), the Reactive Autonomic Component Groups (RACG), as well as Reactive Autonomic System (RAS). This architecture is illustrated in Figure 2.3. In [2] the implementation of autonomic properties is assigned to RAO Leaders (RAOL) at the RAC layer, to RAC Supervisors (RACS) at the RACG layer, and to RACG Manager (RACGM) at the RAS layer. Each layer in this model can only communicate with the layer immediately above or below it. This property provides more modularity for each layer and accords encapsulation and reuse attributes for this model. Using the sample example of Marsworld in Figure 2.2 (Section 2.2), an instance of the RAS meta-model example is represented in Figure 2.4.

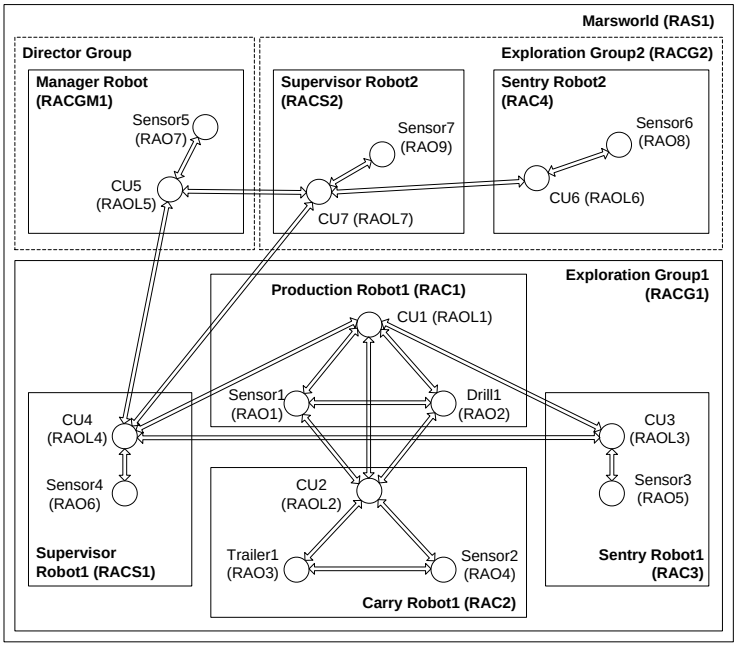


**Figure 2.4:** The RAS model of the Marsworld case study.

Starting from the simplest element in the RAS meta-model, RAO is the atomic member with primary reactive behavior. This element is modeled as a labeled transition system with additional ports, resources, attributes, and logical assertions on those attributes [3]. In the example above, Sensor1 (RAO1) and Drill1 (RAO2) are the RAOs and the control unit CU1 (RAOL1) is the RAOL belonging to Production Robot1 (RAC1). The element immediately about RAO is RAC which is a set of the RAO members. One of the RAOs is the RAOL, which implements proactive behavior whereas the other RAOs exhibit reactive behavior. In this example, Production Robot1 (RAC1) and Carry Robot1 (RAC2) are of this type. The RACG in the higher layer is a group of RACs operating together to perform more elaborate tasks at the group level. RACG is the smallest Reactive Autonomic Element (RAE) in the RAS meta-model that can achieve a complete task in this framework. One of the elements in RACG is a RAC that has

administrative and coordinative tasks in the group and is named as RACS. The Supervisor Robot1 (RACS1) in the example is the supervisor of Exploration Group1 (RACG1). Being a set of RACGs, RAS is the highest layer in this meta-model. One group in RAS is designated as a director group in which one of the RACs is known as the system manager (RACGM). RACGM has the responsibility of managing repositories and coordinating tasks between the groups. Figures 2.5, 2.6, and 2.7 depict the specifications of RAC, RACG and RAS [19].

RAC <name>
  *Members*: <list of the RAO's names in the RAC>
  *Configure*: <list of the pairs of communicating members in the RAC>
  *Leader*: <name of the RAO modeled as a leader for the RAC>
  *Supervisor*: <name of the RACG's supervisor to which the RAC belongs>
  *Neighbors*: <list of the RAC's names that belong to the same RACG>
  *Repository*: <path of the RAC's knowledge base>
End RAC

**Figure 2.5:** Specification of RAC [19].

RACG <name>
  *Members*: <list of the RAC's names in the RACG>
  *Configure*: <list of the pairs of communicating members in the RACG>
  *Supervisor*: <name of the RAO modeled as a supervisor for the RACG>
  *Manager*: <name of the RAS's manager to which the RACG belongs>
  *Neighbors*: <list of the RACG's names that belong to the same RAS>
  *Repository*: <path of the RACG's knowledge base>
End RACG

**Figure 2.6:** Specification of RACG [19].

RAS <name>
  *Members*: <list of the RACG's names in the RAS>
  *Manager*: <name of the RAO modeled as a manager for the RAS>
  *Repository*: <path of the RAS's knowledge base>
End RAS

**Figure 2.7:** Specification of RAS [19].

### 2.3.2. RAS meta-model of behavior

In [3], the reactive behavior of RAO is modeled as a finite state machine augmented with ports, attributes, logical assertions on the attributes and time constraints. The autonomic behaviors of RAOL, RACS and RACGM are modeled as Intelligent Control Loops (ICL) as shown in Figure 2.8 [19].

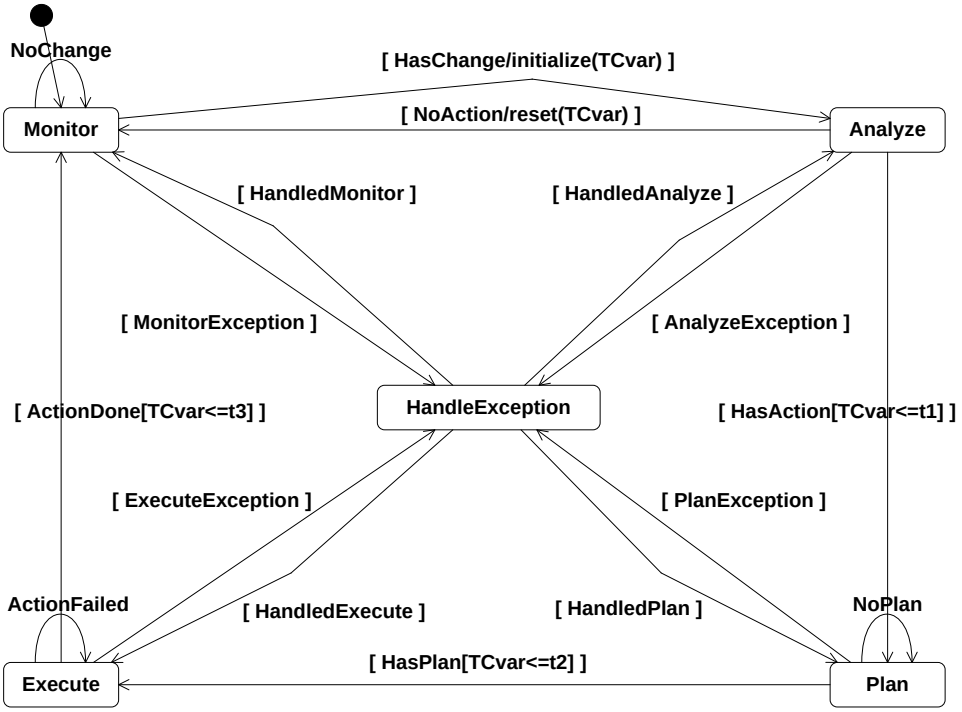


**Figure 2.8:** The Intelligent Control Loop [19].

An ICL consists of *states*, *events* and *transitions*. The state denotes the current status of the component (*Monitor*, *Analyze*, *Plan*, *Execute*, *HandleCondition*). An event triggers a change from one state to another (for example [*HasChange*], [*AnalyzeExeption*]). A transition is a pair of states which specifies the sequence of change triggered by the corresponding event respecting a time constraint (for example *T2*:{*Monitor*, *Analyze*};

[*HasChange*/*initialize*(*TCVar*)]). *T2* is the name of the transition from *Monitor* state to *Analyze* state. *TCVar* is a local clock set to zero when the trigger is received. All timing requirements are specified in terms of *TCVar*. For instance, *[hasPlan]* has to be fired within *12* time units counting from receiving of the request for change.

So far, we stated the static and dynamic aspects of RAS. Since the RAS framework is based on the mathematical Category Theory (CT), in the next section we will introduce the most principal concepts and definitions of CT.

## 2.4. Category Theory

This section introduces the basic notions of Category Theory (CT) required to understand the rest of the material presented in this thesis. Whereas today's complex systems are at most represented by semi-formal diagrams having components and connectors to show interconnections between them, diagrams in CT have a formal intuitive meaning coming from practice. CT is much more complete than being compared to current modeling formalizations of software systems in covering semantics of interconnection, configuration, instantiation, and composition that are important aspects of engineering RAS with autonomic behavior [8].

CT is based on objects and the relationships between them. To make it clear, a category consists of *objects* (*A*, *B*, *C*, etc.) and *morphisms* (*f*: $A \rightarrow B$, *g*: $B \rightarrow C$, etc.). These morphisms, by defining the relationships between the objects, establish a structure for the category. On the other hand, CT provides a set of definitions, techniques, and diagrams that help the system to be examined as a part of a more complex system by building system hierarchies [8]. To understand more the categorical concepts in this

thesis, a few of the CT definitions is discussed below. However, the topic of CT is out of the scope of this thesis and more discussions about the formal definitions are given in [19, 23].

**Definition 2.1**. A category **C** consists of the following data and rules [19]:

- A class of *objects*: *A, B,* etc. We use |**C**| to denote the set of all objects, such as *A*, $B \in$ |**C**|.
- A class of *arrows* (*morphisms*): *f, g*, etc.
- For each arrow $f: A \rightarrow B$, *A* is called the *domain* of *f*, denoted as dom(*f*), and *B* is called the *codomain* of *f*, denoted as cod(*f*). We use **C**(*A*,*B*) to indicate the set of all arrows in **C** from *A* to *B*.
- For each pair of arrows $f: A \rightarrow B$ and $g: B \rightarrow C$, a *composite morphism* is denoted as $g \circ f: A \rightarrow C$.
- For each object *A*, an *identity morphism* has both domain *A* and codomain *A* as $\mathrm{Id}_A: A \rightarrow A$.
- Identity composition: $f \circ \mathrm{Id}_A = f = \mathrm{Id}_B \circ f$ for each morphism $f: A \rightarrow B$.
- Associativity: $h \circ (g \circ f) = (h \circ g) \circ f$ for each set of morphisms $f: A \rightarrow B$, $g: B \rightarrow C$, and $h: C \rightarrow D$.
- Inverse of a morphism $f: A \rightarrow B$ is a morphism $g: B \rightarrow A$ such that $f \circ g = \mathrm{Id}_B$ and $g \circ f = \mathrm{Id}_A$; If *f* has an inverse, it is said to be an *isomorphism*; Also, *A* and *B* are said to be *isomorphic*.

**Definition 2.2**. Let **C** and **D** be categories. **C** is a *subcategory* of **D** denoted as $\mathbf{C} \lhd \mathbf{D}$ if $|\mathbf{C}| \subseteq |\mathbf{D}|$, and the morphisms of **C** are morphisms of **D** as $\mathbf{C}(A_i, A_j) \subseteq \mathbf{D}(A_i, A_j)$ where

$A_i, A_j \in |\mathbf{C}|$; **C** is a *full subcategory* of **D** when $\mathbf{C}(A_i, A_j) = \mathbf{D}(A_i, A_j)$ for all objects of **C** [19].

**Definition 2.3**. A *functor F*: $\mathbf{C} \rightarrow \mathbf{D}$ between two categories **C** and **D** is a mapping of objects to objects and arrows to arrows from **C** to **D** in the following way [19]:

- Object mapping as $F: |\mathbf{C}| \rightarrow |\mathbf{D}|$.
- Arrow mapping as $F: \mathbf{C}(A_i, A_j) \rightarrow \mathbf{D}(F(A_i), F(A_j))$.
- Composition mapping as $F(g \circ f) = F(g) \circ F(f)$ where $g, f \in \mathbf{C}$ and $F(g), F(f) \in \mathbf{D}$.
- Identity mapping: $F(\mathrm{Id}_A) = \mathrm{Id}_{F(A)}$ where $\mathrm{Id}_A \in \mathbf{C}$ and $\mathrm{Id}_{F(A)} \in \mathbf{D}$.

**Definition 2.4**. If **C** is a *full subcategory* of **D** and every $D \in \mathbf{D}$ is isomorphic to some object in **C**, then the insertion functor *F*: $\mathbf{C} \rightarrow \mathbf{D}$ is an *equivalence* [19].

## 2.5. Categorical Specification of the RAS Meta-Model

According to the RAS meta-model, RAC can be specified as a category, say **RAC**. In this category, the objects are RAOs, for example *RAO1*, *RAO2*, etc. The interactions between these RAO members are modeled as morphisms, such as *f*: **RAC** (*RAO1*, *RAO2*). As an example of the Marsworld case study (Section 2.2), a production robot can be considered as a category, say **Production-Robot1** (**PR1)** having the objects *Drill1*, *Sensor1*, *Control-Unit1* (*CU1*) as well as the morphisms **PR1** (*Drill1*, *Sensor1*), **PR1** (*CU1*, *Drill1*), and **PR1** (*CU1*, *Sensor1*). This definition can be extended to the RACG as it can be specified as a category, say **RACG** having objects RAC and their interactions as morphisms *f*: **RACG** ($RAC_i$, $RAC_j$) where $RAC_i$ and $RAC_j$ are objects belonging to **RACG**. RAS itself can be defined as a category, say **RAS** with RACG objects and the

interactions between them as morphisms *f*: **RAS** ($RACG_i$, $RACG_j$) where $RACG_i$ and $RACG_j$ are objects belonging to **RAS**.

The internal and external behavioral specification of RAS is defined with categories **TRANSITION** and **INTERACTION**. As discussed in section 2.3.2, the internal behavior of the RAS framework is modeled by Intelligent Control Loop Model (ICLM). So, this behavioral model can be denoted as category **TRANSITION**, where the objects are sequences of transitions $Seq_1$, $Seq_2$, …, $Seq_n$. For instance, $Seq1 = <Trans_{1\text{-}1}, Trans_{1\text{-}2}, \ldots, Trans_{1\text{-}m}>$, ($n$, $m \geq 1$), and morphisms are isomorphic relations between those sequences. A transition is defined as the tuple (state, event, state). As an example from ICLM (Figure 2.8), a transition can be $Trans_{1\text{-}1}$ = (*Monitor*, *HasChange*, *Analyze*).

For the reason of simplicity, we will refer to RAO, RAOL, RAC, RACS, RACG, and RAGM as Reactive Autonomic Element (RAE). The external interactions of RAS is modeled as category **INTERACTION**, where the objects are the sequences of actions $Seq_1$, $Seq_2$, …, $Seq_n$. For example, $Seq1 = <Act_{1\text{-}1}, Act_{1\text{-}2}, \ldots, Act_{1\text{-}m}>$, ($n$, $m \geq 1$), and the morphisms are isomorphic relations between those objects. The actions are denoted as tuple (*sender*, *TE*, *LE*, *receiver*), stating the sender of TE, Trigger Event (TE) of the action, Last Event (LE) outputted from the action, and receiver of LE. To make it more understandable, an example from the Marsworld case study is: $Act_{1\text{-}1}$ = (*RACS*, *StartRAC*, *HeartbeatRAC*, *RACS*). In this example action, *RACS* sends the triggering *StartRAC* event and receives the *Heartbeat* outputted event in response. For more information about the behavioral categories see [19].

**Definition 2.5**. The interactions of any *RAE* with other *RAE'* or its social life in the category **RAS** is a subcategory of **RAS** denoted as **SOCIAL**(*RAE*), where the objects are

*RAE* and all other *RAE'* $\in$ |**RAS**| which have morphisms with *RAE*, and the morphisms are **RAS**(*RAE, RAE'*) as well as **RAS**(*RAE', RAE*) [19].

The details of applying the Category Theory to prove self-* properties of RAS models can be found in [19].

## 2.6. MAS (BDI) and Jadex

The autonomous characteristics of self-* properties in the RAS model can be refined into multi-agent architecture. The purpose of this thesis is to demonstrate how the RAS model can be implemented with Multi Agent System (MAS) approach. There are many diverse ideas and tools from MAS community to be adapted to implement the RAS model, so it is necessary to have a comprehensive understanding of MAS framework selected for this purpose and the programming tool that will be used to carry out the implementation.

According to [9], an agent is a computer system that can accomplish its task independently on behalf of its user, owner, or certain environment. In comparison to object-oriented model, agents are capable of making decisions and show more autonomous behavior than objects. They can sense the environment by their sensors and act by effectors [9].

A Multi-Agent System (MAS) is a system that consists of a group of autonomous agents working together. These agents are capable of communicating with each other using an Agent Communication Language (ACL). In MAS agents are cooperating with one another to achieve goals of the whole system that is difficult to be reached by each individual agent [9]. The advantages of using MAS are presented in [10, 11, 12]. In [9],

the MAS model is studied from two perspectives: 1) agent interactions; 2) agent architecture.

When the agent interaction is investigated, it is very important to know how the agents prioritize the tasks and resources in relation to other agents. They can have cooperative strategies for interaction or self-interested ones to improve their performance. Also the agents use different protocols and standards to interact with each other. The Agent Communication Language (ACL) defines this facet of multi-agent community [11]. The most popular ACLs are Knowledge Query and Manipulation Language (KQML) and Foundation for Intelligent Physical Agents (FIPA) [10].

The architectural perspective specifies the internal structure of an agent. This structure consists of a set of component models [12] that communicate with each other. Three categories for agent architectures are presented in [9]: 1) deliberative agent architecture; 2) reactive agent architecture; and 3) hybrid agent architecture. One of the main deliberative agent architectures is the Belief-Desire-Intention (BDI) model.

The BDI architecture defines some notations for beliefs, desires, as well as intentions [16]. Beliefs are the knowledge of the agent about itself, other agents, and the environment. Desires indicate the goals to be achieved by the agent. Intentions are what the agent has chosen to do to achieve a goal. Intentions are implemented as executing plans in multi-agent programming environments. The BDI architecture is a model to represent, update and process beliefs, desires, and intentions. Because it is a well-defined structure, agent developers have used the BDI model to provide implementation tools. One of the most widely used agent oriented programming softwares is Jadex that is based on Java Agent Development Framework (JADE) [13].

Jadex is a java-based and FIPA compliant agent development environment. Jadex agents are capable of executing plans as well as sending and receiving messages. These plans can be triggered by external messages from the environment or other agents, or internal goals. The advantages of Jadex as an agent programming tool in comparison with some other similar tools like Jade [44] and Jack [44] according to important criteria relevant for our research are listed in table 2.1.

| Tools / Criteria | Jadex | Jade | Jack |
|---|---|---|---|
| Simplicity | X | X | X |
| Java-based | X | X | X |
| XML support | X | — | — |
| BDI-based | X | — | X |
| Eclipse plug-in | X | X | X |

**Table 2.1:** The advantages of Jadex.

The main concepts in Jadex are beliefs, goals, plans and messages. Beliefs in Jadex are stored as a database in the belief base. This database consists of a set of beliefs that make up the knowledge of the agent. The content of a belief in Jadex is a value known as a fact. Jadex also provides beliefs having a set of facts. This belief base can be updated during the execution of a plan. Jadex takes advantage of an Object Query Laguage (OQL) that looks like a query language (adapted from object-relational database world) [13] to create more complex select queries to search belief base.

Goals are one of the most important motivational forces for Jadex agents to take action. An agent will keep up with a set of tasks for its goals until it assesses the goal as being reached, unreachable, or not desired anymore [13]. In Jadex, there are four kinds of

goals: 1) perform goal: states that some action should be done but any specific result is not expected; 2) achieve goal: a target is determined and the goal is to attain that target state; 3) query goal: is used to enquire information about something; and 4) maintain goal: is applied to preserve a state in its desired condition.

Plans are the agents' method and blueprint to perform their tasks. In fact a plan is what an agent executes in response to an internal or external trigger. In Jadex, a plan has two parts. The first part is the plan header that determines the conditions that trigger the plan. The second part is a Java class inherited from Plan class that overrides its *body()* method that is run when the plan is triggered.

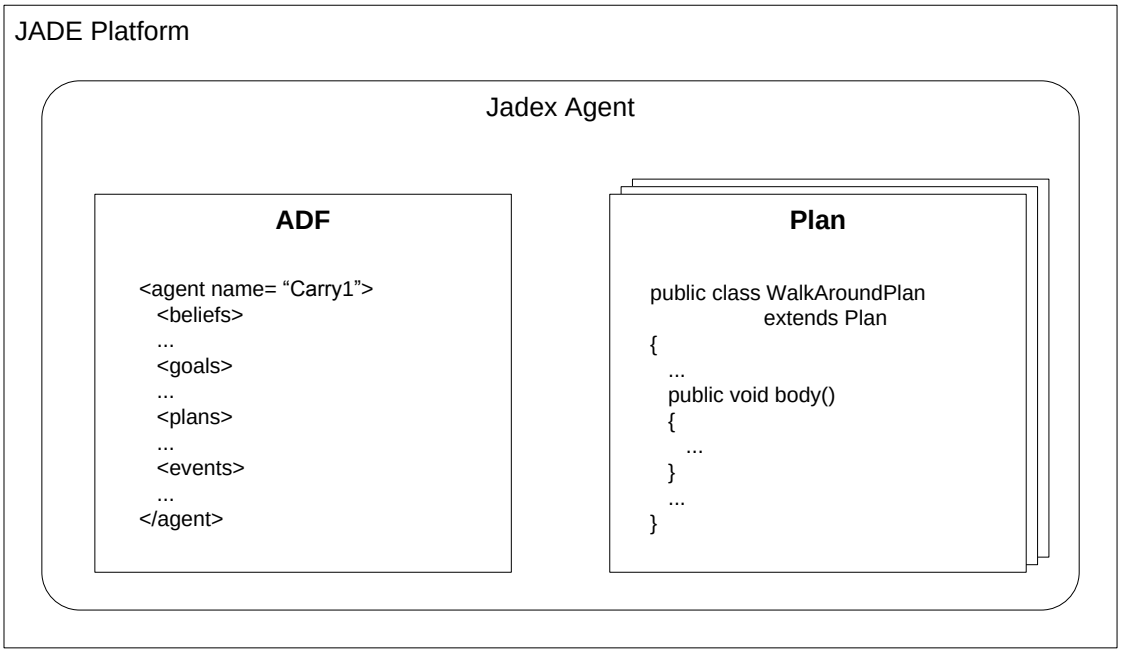


**Figure 2.9:** The structure of a Jadex agent [13].

In Jadex an agent is defined in an XML file known as Agent Definition File (ADF). The XML tags in this file specify beliefs, goals, events, plans, and all other elements necessary for the agent definition. In fact all agents in Jadex are instantiated from the

ADF just like objects that are instantiated from class definitions. It is possible also to declare the initial state of an agent in ADF using configuration tag like the initial beliefs, initial goals, and initial plans. Plans in Jadex are the Java code each of which stored in a Java class file. The name of these plans must correspond to plan header definition in the ADF. Figure 2.9 illustrates an agent definition in Jadex and one of its plan files. Finally, Jadex comes with the Jadex Control Center (JCC) that is used to load and run Jadex agents [13].

## 2.7. Fault-Tolerance

As discussed in [14], the most essential property of autonomic systems is self-management that consists of: 1) self-configuration: the ability of adapting to the changing environment; 2) self-healing: the capability of detecting and resolving the problems; 3) self-optimization: the ability of tuning the resources; and 4) self-protection: the ability of self-defending against any damage. In this research, the focus is on self-healing property and fault-tolerance as a mechanism for achieving self-healing in RAS.

The self-healing property denotes that an autonomic system is capable of finding (detecting) the faults. The autonomic system has the ability to analyze the problem using error log files or state snapshots. Using this knowledge, the autonomic system takes appropriate action to recover itself if possible or request human intervention in case of necessity [14].

Fault-tolerance is defined as the property that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components. Fault-tolerance is a sub-quality of Reliability according to the ISO 9126.

There are discussions about the similarities and differences between fault-tolerance and self-healing [17]. Fault-tolerance is an existing area that has proven to be effective at the later stages of design. In other words, fault-tolerance is an operationalization of the self-healing property, which explains how to provide, by redundancy, service complying with the specification in spite of faults having occurred or occurring [18].

Since in the RAS model there is a group of autonomic elements which work together, the fault can happen in any of the RAE components. Thus, fault-tolerance can be regarded as the recovery of a crashed element. According to the specifications of swarm systems [19] (a large number of inexpensive robots), if the crashed robot could be replaced with a similar available robot, the whole system must be able to continue to function properly. This property is called substitutability of RAE [19].

The substitutability property of an RAE denotes that the RAE can be replaced by another RAE' if and only if 1) they belong to the same type (RAO, RAOL, RAC, RACS, RACG, or RACGM); 2) they have equivalent social lives as **SOCIAL**(RAE)~**SOCIAL**(RAE'); 3) when they are regarded as two categories, they have equivalent internal structure as **CAT**(RAE)~**CAT**(RAE'); 4) their internal as well as interactive behavior that is regarded as the following two categories, is equivalent as **TRANSITION**(RAE)~**TRANSITION**(RAE') and **INTERACTION**(RAE)~ **INTERACTION**(RAE').

The authors in [19] use some scenarios from Marsworld case study to illustrate the substitutability property of different components in the RAS meta-model. These scenarios simulate the crash of a component in the RAS meta-model and show how the substitutability property is used to replace the damaged component to achieve fault-

tolerance in the system. The implementation of fault-tolerance with Jadex for the Marsworld case study to prove the substitutability property is discussed in Chapter 4.

## 2.8. Model Transformation

Since it has been proven that substitutability property of RAE fulfills the fault-tolerance of the RAS meta-model and considering similarities between reactive autonomic system and multi-agent community, this thesis proposes a model transformation framework to transform the RAS model to MAS. To carry out the transformation process, many model transformation approaches with an extend domain of methods and tools are available. In this section, we will not try to go in deep into different model transformation methods, but on the other hand we will identify an approach that is appropriate and convenient for our purpose.

A model transformation in model-driven engineering [21] takes as input a model conforming to a given input meta-model and produces as output another model conforming to a given output meta-model. There are many tools that support the automation of model transformation. These model transformation development tools not only offer the possibility of applying predefined model transformations on demand, but also offer a language that allows (advanced) users to define their own transformation rules and execute them.

Performing a model transformation, i.e., taking one or more models as input and producing one or more models as output, requires a clear understanding of the abstract syntax and semantics of both the source and target. A common technique for defining the abstract syntax of models and the inter-relationships between model elements is meta-

modeling. To define the required meta-models, we need an appropriate data schema to express input and output models. There are some tools that use graphical schema to define their source and target specifications, e.g., DOME and GME2000 [21]. Also Unified Modeling Language (UML) is applied as a meta-model in a large number of tools such as Objecteering, RationalRose, and Together [21]. As stated in [20], these tools offer three transformation approaches for their users: the direct model manipulation approach, the intermediate representation approach, or the transformation language support approach [21].

XML (Extensible Markup Language) is specially designed to be easy to use over the Web, to be human-readable and straightforward for applications to read and understand. XML is quickly becoming the universal syntax for information transfer; therefore a vast amount of information transformation uses XML as the input and/or output data format.

Since our input and output models are serialized in XML format using the XML meta-data, implementing model transformations using Extensible Stylesheet Language Transformation (XSLT), which is a standard technology for transforming XML, seems very attractive. XSLT is an XML-tool to perform model transformation. It defines the mapping from XML into another markup language such as XML, HTML, or into plain text. XSLT stylesheets are interpreted by XSLT processors, which generate a result from source XML document. XSLT processors can be embedded in web browsers or be executed from the command line to run stylesheets [45]. Figure 2.10 depicts very simply the transformation process.

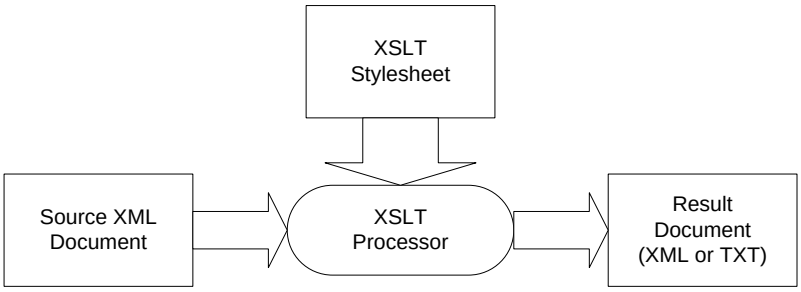


**Figure 2.10:** The transformation process in XSLT.

XSLT uses XPath to select parts of XML to process and to perform calculations. XPath, the XML Path Language, is a query language for selecting nodes from an XML document. The most important role of XPath is to collect information from an XML document by navigating through the document. A secondary role of XPath is as a general expression language, to perform calculations.

## 2.9. Conclusion

In this chapter, we discussed the RAS framework that proposes a formal model for specifying and verifying structure and behavior of RAS based on the mathematical category theory (see Figure 2.1). We illustrated how the different architectural elements of RASF as well as the behavioral prototype can be represented by this theory. Besides, we discussed how category theory can be used to model the self-healing property of RASF using substitutability property of the RAS components. In the next step, we showed that the multi-agent system is an appropriate solution to implement RAS. Finally, we discussed a model transformation blueprint using XSLT to transform RAS to MAS with regard to self-healing property.

In the next chapter, we will explain the details of the transformation process from RAS to MAS. The grammar that produces the input meta-model to the transformation engine will be presented. The input XML meta-model and the output meta-model in Jadex will be explained and finally the transformation rules that take the input model and produce the output model will be introduced.

# Chapter 3: Transformation from RAS to MAS

## 3.1. Introduction

In this chapter, the transformation method from the RAS model to MAS model is discussed. To do this, the input and output file format will be investigated. The input meta-model of this transformation process is created using a grammar defined from the RAS architecture. The result of this grammar definition is an XML file that represents each type of the RAS elements. In fact, a set of transformation rules will be executed on this XML format to create the output model in Jadex, which is a Java-based MAS-BDI compatible agent programming tool. The output model in Jadex consists of Agent Definition Files (ADF) in XML format, which define beliefs, goals, message events and plan headers as well as the plan files in Java code that contain the body of executable plans. Figure 3.1 illustrates the transformation process.

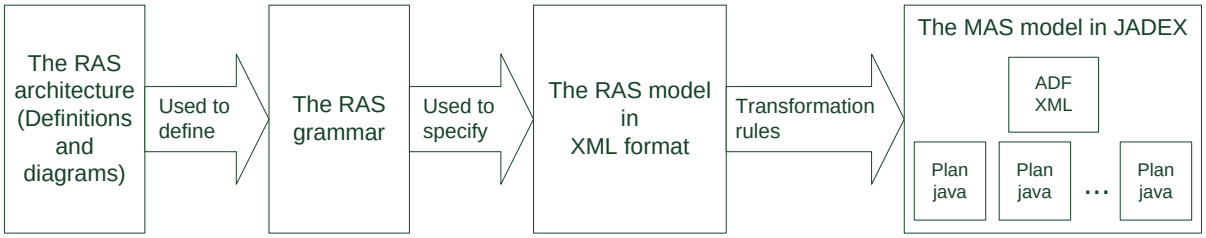


**Figure 3.1:** Transformation process from RAS to MAS.

The first step in this transformation process, i.e. creating the RAS model in XML format from the RAS grammar, is done using the RAS definitions and diagrams (from the RAS architecture). The second step, i.e., defining transformation rules, is implemented using Extensible Stylesheet Language Transformation (XSLT) [45], which is a Model Transformation framework for XML format and its language XPath. The transformation rules will be analyzed from two angles, static architecture and dynamic behavior. The dynamic behavior will be focused on one self-* property, namely self-healing behavior of the RAS model using the sequence diagrams.

## 3.2. The RAS Grammar

In this section, the RAS grammar is illustrated. This grammar defines the RAS concepts including Reactive Autonomic Object (RAO), RAO Leader (RAOL), Reactive Autonomic Component (RAC), RAC Group (RACG), Group Manager (GM), Group Supervisor (GS) and Reactive Autonomic System (RAS) based on Extended BNF ISO 14977 standard [24]. In this standard, the notation "{}-" means "one or more".

```
RAOL = RAO, repository;
repository = {property}-;
property = name, type, {value}-;
RAC = RAOL, {RAO}-;
GM = RAC, RAS_repository;
GS = RAC, RACG_repository;
RACG = GS, {RAC}-;
RAS = GM, {RACG}-;
```

**Figure 3.2:** The RAS static grammar.

Figure 3.2 shows the grammar for the RAS static architecture. This grammar uses regular expressions to define constants and operations. In the RAS architecture, RAO is

the atomic element that cannot be broken down. Following is the description of each of the regular expressions.

*RAOL = RAO, repository*; RAOL is a RAO that is the leader of other RAOs inside the RAC. The leadership here means that RAOL contains a repository that stores persistent knowledge of RAC.

*repository = {property}-*; The repository consists of one or more properties. This can be interpreted as a database for RAC.

*property = name, type, {value}-*; The property is a triple consisting of the name of the property, its type and one or many values that can be assigned to it. In fact the properties are the pieces of information inside the repository.

*RAC = RAOL, {RAO}-*; A RAC consists of one RAOL and one or many RAOs. These RAOs are communicating with each other and with the leader.

*GM = RAC, RAS_repository*; A GM is an intelligent RAC having a repository that serves to store knowledge of different RACG groups in RAS.

*GS = RAC, RACG_repository*; A GS is an intelligent RAC having a repository that serves to store knowledge of different members in a RACG group.

*RACG = GS, {RAC}-*; A RACG consists of a GS that is itself a RAC and one or more RACs.

*RAS = GM, {RACG}-*; A RAS captures the whole system that consists of a GM that is in fact a RAC and one or more RACGs.

The behaviors of the RAO and RAC components are illustrated by the Extended BNF grammar shown in Figures 3.3 (a) and 3.3 (b). In fact, in the RAS model RAO is the atomic element having an atomic reactive behavior that consists of a trigger and the

corresponding plan to deal with it. The RAS model has also a proactive behavior that consists of a set of execution paths. Following is a short description for each of the regular expressions in Figures 3.3 (a) and 3.3 (b).

```
RAO-behavior = {reactive-atomic}-;
reactive-atomic = reactive-trigger, message;
reactive-trigger = sender, event, receiver;
message = sender, event, receiver;
sender, receiver = RAO | RAOL | GM | GS | ENV;
event = EO | EI | IN | timeout;
timeout = integer;
```

**Figure 3.3 (a):** The RAO behavioral grammar.

```
RAC-behavior = {reactive | self-properties}-;
reactive = {ex-path}-;
ex-path = reactive-trigger | proactive-trigger, {message};
sender, receiver = RAO | RAOL | GM | GS | ENV;
proactive-trigger = sender, IN, receiver;
reactive-trigger = sender, event, receiver;
message = sender, event, receiver;
self-properties = {goal}-;
goal = name, ex-path;
```

**Figure 3.3 (b):** The RAC behavioral grammar.

*RAO-behavior = {reactive-atomic}-*; A RAO is the atomic element in the RAS model and his behavior consists of one or more reactive-atomic behaviors.

*reactive-atomic = reactive-trigger, message*; A reactive-atomic behavior is defined as a trigger of a certain event and a response to that trigger.

*reactive-trigger = sender, event, receiver*; A reactive-trigger consists of a sender of the trigger, for instance another RAO, an event that represents the trigger itself, and the receiver of the trigger.

*sender, receiver = RAO | RAOL | GM | GS | ENV*; The sender and the receiver of the message or trigger can be one of the elements mentioned above.

*event = EO | EI | ON | timeout*; An event can be of type EO (External Output), EI (External Input), IN (Internal) or an integer value that represents the timeout of a message. In this case no message is sent but this value determines how long the sender of the message must wait before issuing a timeout exception.

*timeout = integer*; As mentioned above, timeout is an integer value that denotes the duration before taking action when there is no response message.

*proactive-trigger = sender, IN, receiver*; Proactive-trigger is similar to reactive-trigger with event replaced by internal event.

*ex-path = reactive-trigger | proactive-trigger, {message}-*; An ex-path (execution path) consists of a trigger and one or more messages.

*RAC-behavior = {reactive | self-properties}*; The behavior of the RAC element is the union of reactive and self-properties behavior.

*reactive = {ex-path}-*; The reactive behavior of an element consists of one or more execution paths.

*self-properties = {goal}-*; The self-properties of the RAS element consists of one or more goals.

*goal = name, ex-path*; A goal is a combination of name representing the name of the goal and an execution path to achieve the goal.

The above grammar will be used to produce XML files denoting each RAS element such as RAO and RAC. This process is done using the grammar expressions as a conceptual source.

## 3.3. Input Model

The input model to the transformation process is the RAS architecture captured and represented in XML format. This section will discuss this input XML file and its *tags* and *attributes* to better understand this input model. The input model defines the RAS framework from two points of views: static view and dynamic view. All the elements of

the RAS framework that consists of *RAOs*, *RACs*, *RACGs* and *RAS* are specified in XML format. RAOs are considered as atomic elements from architectural point of view. On the other hand, the behavioral structure of RAOs is specified in XML files as atomic behaviors that will be discussed later in this chapter. The section starts with the investigation of RAO and RAC that are the most important elements and then moves to the other elements to completely clarify the input structure.

### 3.3.1. RAO Specification

This element is the atomic element of the model and its behavior is assumed to be atomic. Figure 3.4 illustrates the XML specification of RAO.

```
<RAO name = "rao-name">
   <REACTIVE-ATOMIC>
      <TRIGGER name= "trigger-name"/>
      <PLAN name= "plan-name"/>
      <RESPONSE name= "response-name"/>
   </REACTIVE-ATOMIC>
</RAO>
```

**Figure 3.4:** The RAO definition in XML format.

For RAO there is no architectural definition since it is an atomic element. On the other hand, its reactive behavior is specified using trigger-response pairs, which capture the atomic behavior of RAO. Following is the description of the tags in Figure 3.4.

The <RAO name = "*rao-name*"> </RAO> pair specifies the beginning and end of a RAO definition in XML format. The first tag has a *name* attribute that defines the name of the RAO.

Each RAO may have one or many <REACTIVE-ATOMIC> tags, and each one of them specifies one atomic behavior of the RAO. This tag has one <TRIGGER>, one <PLAN> and one <RESPONSE> sub-tag.

The <TRIGGER> tag specifies the name of the message event that triggers the first action, which is executed in a plan body. The name of the trigger is specified in the *name* attribute of this tag.

The <PLAN> tag determines the name of the plan to be executed when receiving the trigger. Executing this plan allows performing the atomic tasks and preparing the response message event. The *name* attribute in this tag defines the name of the plan that corresponds to an executable program.

The <RESPONSE> sub-tag determines the response message event to be sent in response to the trigger received by the RAO. This response is created and sent in the plan body that is executed by the trigger. This tag has a *name* attribute that identifies the message event to be sent.

### 3.3.2. RAC Specification

RAC is the principal element of the model and consists of atomic elements RAOs with autonomic behavior. The XML specification of RAC consists of *tags* that define the static structure of this element and other *tags* that determine its behavior. Figure 3.5 shows the XML format defining RAC.

3.3.2.1. *Static view*:

The <RAC></RAC> tag surrounds all other tags of RAC and it has a *name* attribute that specifies the name of the RAC and can be any name of type *string*. This name is important because it is used by other model elements to refer to this element.

```
<RAC name = "rac-name">
   <MEMBERS>
      <MEMBER name = "rao-name"/>
   </MEMBERS>
   <INTERACTIONS>
      <INTERACTION source = "source-rao" name = "event-name" target = "target-rao"/>
   </INTERACTIONS>
   <REACTIVE-BEH>
      <LIST-EX-PATH>
      <EX-PATH name = "ex-path-name">
         <TRIGGER>
            <SENDER name = "environment"/>
            <EVENT name = "trigger-name"/>
            <RECEIVER name = "receiver-name"/>
         </TRIGGER>
         <MESSAGE>
            <SENDER name = "sender-name"/>
            <EVENT name = "event-name" type= "event-type">
               <TIMEOUT min= integer max = integer/>
            </EVENT>
            <RECEIVER name = "receiver-name"/>
         </MESSAGE>
      </EX-PATH>
      </LIST-EX-PATH>
   </REACTIVE-BEH>
   <SELF-PROP>
      <GOAL name = "goal-name" path = "ex-path-name">
         <EX-PATH>
            ...
         </EX-PATH>
      </GOAL>
   </SELF-PROP>
   <LEADER name = "raol-name"/>
   <REPOSITORY>
      <PROPERTY name="property-name" type="property-type">value</PROPERTY>
   </ REPOSITORY>
</RAC>
```

**Figure 3.5:** The RAC definition in XML format.

The <MEMBERS> </MEMBERS> tag group specifies all RAOs that belong to this RAC. Under this tag, for each RAO a <MEMBER name = "*rao-name*"/> is added.

The <INTERACTIONS> </INTERACTIONS> tag group defines the existence of any interaction between the specified RAOs. This information does not represent any dynamic feature of the model as it does not capture any behavioral aspect. As a matter of fact, only the communication structure of RAOs is captured in this tag. For each connection between two RAOs (for instance RAO1 and RAO2) one <INTERACTION source = "*RAO1*" target = "*RAO2*"/> will be created under <INTERACTIONS> tag.

The <LEADER name = "*rao-name*"/> tag specifies the RAO leader (RAOL) among the group. The *name* attribute determines the name of this RAOL in the RAC.

The <SUPERVISOR name = "*rac-name*"/> tag defines the supervisor of the RAC in the RACG group containing this RAC.

The <NEIGHBOURS> </NEIGHBOURS> tag group specifies the neighbor RACs that the current RAC can communicate with. For each neighbor RAC a <NEIGHBOUR name = "*rac-name*"/> sub-tag is added under <NEIGHBOURS> tag.

The <REPOSITORY> </REPOSITORY> tag group specifies the knowledge inside a RAC. This knowledge can consist of any information about different properties. Inside this tag there is one sub-tag for each property. In the group <PROPERTY name = "*property-name*" type = "*property-type*">*value*</PROPERTY> the *name* attribute specifies the name of the property, the *type* attribute specifies the data type of the property and the *value* content determines the value of the property.

3.3.2.2. *Dynamic behavior*:

The dynamic behavior is captured by two principal tags inside the RAC specification. The <REACTIVE-BEH> and <SELF-PROP> tags. The <REACTIVE-BEH> tag specifies the reactive behavior of the RAC element. The <SELF-PROP> tag determines the self-* properties of the RAC element.

The <REACTIVE-BEH> </REACTIVE-BEH> tag group consists of one or more <EX-PATH> tags each of which corresponds to one execution path of the RAC. These execution paths are captured from the sequence diagrams that determine the behavior of the element (see Figures 3.14 and 3.15 in this chapter).

The <EX-PATH name = "*ex-path-name*"> </EX-PATH> tag group determines one execution path of the RAC element and consists of one triggering event, <TRIGGER>, and a sequence of one or more messages, <MESSAGE>.

The <TRIGGER name = "*trigger-name*"/> tag is an external event recognized by the RAC element that triggers an event sequence. In fact it is the starting point of the execution path in the RAC element.

The <MESSAGE> </MESSAGE> tag group defines one message sequence inside the execution path. Each <MESSAGE> tag consists of one sender, one event and one receiver of the event message.

The <SENDER name = "*sender-rao*"/> tag specifies the sender RAO of the event message. It has a *name* attribute that represents the sender's name.

The <EVENT name = "*event-name*" type = "*event-type*"/> tag designates the event message that is sent by the RAO. This tag has a *name* property specifying the name of the event, and a *type* attribute defining the type of the event message.

The <RECEIVER name = "*receiver-rao*"/> tag defines the receiver RAO of the event message. This tag has a *name* attribute that determines the receiver's name.

What is described till now about the behavioral model of RAC is the reactive behavior. In other words, this behavior is the reaction of the RAO to the incoming events from its environment or other RAOs. What will be discussed next is the definition of the proactive behavior of RAC. In fact the triggering of the proactive behavior is a goal internally defined inside the RAC component. For the RAS model this behavioral model corresponds to the self-* properties of RAC such as self-tolerance, self-optimization, self-configuration, self-protection, etc.

The proactive behavior of the RAC element is defined by <SELF-PROP> </SELF-PROP> tag group. This tag consists of one or more goal tags. The RAC element has to achieve all of these goals to fulfill the specified self-* property.

The <GOAL name = "*goal-name*" path = "*ex-path-name*"/> tag has a *name* attribute and a *path* attribute. The *name* attribute specifies the name of the goal and the *path* attribute defines the execution path to be followed to achieve the goal. The execution path can be specified by just mentioning an *ex-path-name* that has been defined before or by directly specifying the execution path using a <EX-PATH> </EX-PATH> as a sub-tag inside the goal tag.

### 3.3.3. RACG Specification

According to the grammar given in Figure 3.2, RACG consists of one or more RACs grouped together to achieve more complex goals. From the architecture and behavior perspectives, RACG is very similar to RAC. As can be seen in Figure 3.6, all the tags of RACG are the same as for RAC. The main difference is that RACG is one level higher in the hierarchy structure of RASF. In fact in a RAC the member elements are RAOs, but in a RACG the members are RACs. Another difference between RAC and RACG is that, in <INTERACTIONS> tag of RACG the *source* and the *target* attributes are RAC names. It is very important to know that the interaction between RACs in RACG is achieved via the RAOL element of the RAC components. In other words, the RAC itself is a group of RAOs that communicate with other RACs through its RAOL. In the transformation process, the RAC will be transformed to a group of agents such that each of them is replacing its composing RAOs. The leader in RACG definition is called here a *supervisor* and in fact is the most intelligent RAC in the group.

```
<RACG name = "racg-name">
   <MEMBERS>
      <MEMBER name = "rac-name"/>
   </MEMBERS>
   <INTERACTIONS>
      <INTERACTION source = "source-rac" name = "event-name" target = "target-rac"/>
   </INTERACTIONS>
   <LEADER name = "supervisor-name"/>
   <REPOSITORY>
      <PROPERTY name="property-name" type="property-type">value</PROPERTY>
   </ REPOSITORY>
</RACG>
```

**Figure 3.6:** The RACG definition in XML format.

The behavioral tags of RACG are exactly the same as for RAC. The only difference is that the communications between RACs are done through their RAOLs.

## 3.4. Output Model

The output model is the MAS framework in BDI architecture defined and implemented in Jadex. What is discussed so far is the input model serialized in XML format. To better understand and develop the transformation rules, we discuss the output model and the facilities it provides to specify the static architecture and dynamic behavior.

The target model of this transformation is a BDI-based MAS model in Jadex; a Java-based multi agent platform. In this platform, the agents are defined by two file formats. The definition of the agent in XML format that is stored in a file called ADF (Agent Definition File) and the body of plans in the Java language format stored in text files having Java extension.

The structure of Jadex agents is stored in ADF XML files. An ADF file consists of different *tags* to implement various concepts of the BDI model. This section discusses the most important tags that take part in our transformation process. Figure 3.7 introduces briefly the structure of an ADF file.

The <agent> tag defines the header of the ADF file consisting of the name of the agent, the version of the xml schema and also the package declaration. What is specified in the package declaration is the location of the agent's class files for beliefs and plans that would be searched for the first time. The ADF file is located in the package folder as mentioned in this tag attribute.

The <imports> tag is used to specify the list of library packages that contain different class definitions used in various sections of the ADF file. This tag is similar to the *imports* command in the Java programming language.

The <capabilities> tag defines the capabilities included in the agent. Each capability has its own ADF definition file, and also the plan body files but must be included inside an agent to be useful. In fact when a capability is added to an agent using <capabilities> tag, all the defined beliefs, goals, plans and events are available for the host agent. Generally speaking, capabilities can be regarded as libraries captured in XML format and Java programs conformed to Jadex so that any agent can have access to it.

```
<agent name = "agent_name" package = "package_name" >
   <imports>
      <import>jadex.*</import>
   </imports>
   <capabilities>
      <capability name = "amscap" file = "jadex.planlib.AMS"/>
   </capabilities>
   <beliefs>
      <belief name = "belief_name" class = "class_name">
         <fact></fact>
      </belief>
   </beliefs>
   <goals>
      <achievegoal name = "goal_name">
         <dropcondition>condition</dropcondition>
      </achievegoal>
   </goals>
   <plans>
      <plan name = "plan_name">
         <body class = "java_class_name"/>
         <trigger>
            <goal ref = "goal_name"/>
         </trigger>
      </plan>
   </plans>
   <events>
      <messageevent name = "event_name" type = "fipa" direction = "direction">
```

```
        </messageevent>
    </events>
    <configurations>
        <configuration name = "configuration_name" >
            <plans>
                <initialplan ref = "initial_plan_name"/>
            </plans>
        </configuration>
    </configurations>
</agent>
```

**Figure 3.7:** Jadex Agent Definition File in XML format.

The <beliefs> tag specifies the knowledge of an agent in Jadex BDI model. Inside the <beliefs> tag there are two types of belief tags, namely, <belief> tag for the single valued beliefs and <beliefset> tag for the multi-valued beliefs. The <belief name = "*belief-name*" type = "*belief-type*"> of <beliefset …> has a *name* attribute that specifies the name of the belief or belief set and a *type* attribute that determines the data type of the belief or belief set. Under these two tags the facts are defined using the <fact> or <facts> tags. For the <belief> tag there is just one <fact> tag and for the <beliefset> tag there is more than one <fact> tag or one <facts> tag. The content of the <fact> or <facts> tag is the value of the tag. For example, if the value of the fact is the name of a city (for instance Quebec), it is defined as <fact>"*Quebec*"</fact>.

The <goals> tag determines the goals definition of the agent. Goals are one of the principal components in the Jadex BDI model. There are four different types of goals supported in Jadex: perform goals, achieve goals, query goals, and maintain goals. For the description of each type of goal please refer to Section 2.6 in Chapter 2. The achieve goals are denoted by the tag <achievegaol>, the perform goals by <performgoal>, the

query goals by <querygoal>, and the maintain goals by <maintaingoal> tag. The first two types of goals are the most important ones in our transformation process.

The <achievegoal name = "*goal-name*"> tag has an attribute *name* that specifies the name of the goal. Each goal can have one or many parameters that can be passed to it during its creation cycle. These parameters are specified by <parameters name = "*p-name*" class = "*c-name*"> tag that has a *name* and a *class* attribute. Also an <achievegoal> tag can have a <creationcondition> tag to specify the creation condition of the goal, a <contextcondition> tag to specify the condition that must hold to keep the goal active, a <dropcondition> tag to specify the drop condition of the goal, a <targetcondition> tag to specify the target condition of the goal, and a <deliberation> tag to inhibit other goals from execution.

The <performgoal name = "*goal-name*" retry = "*true*" exclude = "*never*"> tag has an attribute *name* that defines the name of the goal. Two other important attributes are *retry* and *exclude*. The *retry* attribute means this goal will be repeated again when terminated and the *exclude* attribute determines the plans to be excluded after each execution. Similar to achieve goals, the tags <creationcondition>, <contextconditon>, <dropcondition>, and <targetconditon> can be added under <performgoal> tag to specify different kinds of conditions for the goal. Also the <deliberation> tag is used to define the inhibition of other goals.

The <querygoal name = "*goal-name*"> tag defines the query goals and has an attribute *name*. This type of goal is similar to the previous two goals with a small difference in dealing with the output parameters.

The <maintaingoal name = "*goal-name*"> tag specifies the maintain goals. Like the previous three goal types it has also a *name* attribute to determine the name of the goal. Generally speaking, this goal differs from the other types in the way it is activated and terminated. Under the <maintaingoal> tag there is a <maintaincondition> sub-tag that specifies the condition that is necessary to maintain a situation. This goal also has the <dropconditon> tag to define the condition that terminates the goal.

The <plans> tag defines the header part of the agent's plans. This tag has one or more <plan> tags each of which declares a plan header for the agent.

The <plan name = "*plan-name*" priority = *0*> tag identifies a plan using two attributes, *name* and *priority*. The *name* attribute is required to refer to the plan but the *priority* attribute is optional. The most important sub-tags in <plan> tag are the <body> and <trigger> tags.

The <body> tag specifies the plan body Java class that is instantiated when the plan is triggered. For example if there is a plan named *ping* that corresponds to the Java class name *PingPlan()*, the body tag will be defined having the instantiation statement of the Java class in its content part like <body> new PingPlan() </body>.

The <trigger> tag determines the triggering component of the plan that can be an event or a goal. Depending on what triggers the plan, the sub-tag of <trigger> would be a <messageevent> tag or a <goal> tag. For example if a message event named *query-ping* is triggering a plan, there will be a tag <messageevent ref = "*query-ping*"> tag under <trigger> tag in the plan. There are also some additional triggering tags such as the

<condition> tag to specify a precondition of the plan, the <beliefchange> and <beliefsetchange> tags to specify a trigger of a change of a belief or set of beliefs, and the <factadded> and <factremoved> tags to specify a trigger initiated from adding a fact or removing it from a belief set.

The <events> tag defines the different events that the agent can react to. In Jadex there are two types of events. The internal events represented by the <internalevent> tag and the message events declared by the <messageevent> tag.

The <internalevent name = "event-name"> tag specifies the internal events of an agent and has an attribute *name*. This type of events is used in an agent to transfer an internal message to all plans in which this event is specified as a triggering event. The internal event can have a parameter tag having two attributes *name* and *class* like <parameter name = "*parm-name*" class = "*parm-type*">.

The <messageevent name = "*msg-name*" type = "*msg-type*" direction = "*direction*"> tag specifies the messages sent from and received by the agent. The *name* attribute defines the name of the message. The *type* attribute denotes the standard type of the message to be transferred. In this case, the *fipa* standard is always used. The *direction* attribute declares whether the agent is sending or receiving this message. The <messageevent> tag can have <parameter> sub-tags. These parameter sub-tags could be of various names and contents and will be described in the next section according to each specific use.

## 3.5. Transformation rules

This section describes the rules to transform the input model called Left Hand Side (LHS) (the RAS meta-model) to the output model called Right Hand Side (RHS) (the MAS meta-model). The two previous sections dealt with the input and output models and the format of two meta-models in XML format. This section will investigate the rules to map each input component to its corresponding component in the output.

To better understand the transformation rules, this section will use a special notation that fits the algorithms that are based on XSLT and XPath. XSLT and XPath traverse the input XML file from top to bottom and by visiting each tag it creates the output model. The transformation rules for the static view (architecture) and dynamic view (behavior) will be investigated in two different sections.

3.5.1. *Static view transformation*:

**R1: <RAC> to <package> rule:** Jadex is a Java based environment including all the concepts related to the MAS model. The concept of packages can be used to create the RAC encapsulation. The *name* attribute of the <RAC> tag will serve to create a package for the agents that belong to it.

**R2: <MEMBERS> to <package> rule:** The <MEMBERS> tag that contains <RAO> tags will determine all the RAOs that belong to one RAC. In fact, the <RAC> tag declares the name of the package and the <MEMBERS> tag tells what agents belong to this RAC.

**R3: <RAO> to <agent> rule:** The atomic RAO component of RAS model is transformed to the <agent> tag with all its attributes. These attributes consist of *name*, *package*, *xmlns*, *xmlns:xsi* and *xsi*. The package attribute as mentioned before will be determined from RAC tag's name attribute.

**R4: <INTERACTION> to <messageevent> rule:** The <INTERACTION> tag will specify the messages that can be sent between agents. This tag has three attributes including the *sender*, *event-name* and *receiver*. The *sender* attribute is a RAO name that has been transformed to an agent. As a result, in the transformation process, a message event having *event-name* name will be created in the sender agent with the *direction* attribute valued "send". Also the same message event will be created in the receiver agent but with the *direction* attribute having the value "receive". Other attributes and also <parameter> sub-tags in the <messageevent> tag will have different values in different cases and environments. For example in our case study (see Chapter 4), the *type* attribute of <messageevent> tag has the value "fipa" and according to the type of the message, the content of <parameter name = "*performative*"> tag can be SFipa.INFORM, SFipa.REQUEST, etc.

**R5: <LEADER> to <beliefs> rule:** The <LEADER> tag can serve to determine the agent inside a RAC that will contain the repository of the RAC. In fact, all the knowledge of the RAC will be concentrated inside the leader agent by creating the <beliefs> tag containing the agent properties.

**R6: <REPOSITORY> to <beliefs> rule:** The <REPOSITORY> tag will be transformed into <beliefs> tag inside the RAOL agent. As mentioned before, the <LEADER> tag will

specify the name of the leader agent. The property tags under this tag will be transformed to <belief> or <beliefset> tags.

**R7: <PROPERTY> to <belief> or <beliefset> rule:** The <PROPERTY> tag is transformed to <belief> tag for single valued properties, and to <beliefset> tag for multi valued properties. The *name* attribute becomes the name of the belief, the *type* attribute becomes the type of the belief and the *value* content is transformed to <fact> sub-tags with the same value.

3.5.2. *Dynamic view transformation*:

The dynamic behavior of the RAS model is developed using the sequence diagrams for each of the self-* properties. These sequence diagrams show possible scenarios of self-* properties for each of the RAS elements. The transformation process takes advantage of these sequence diagrams captured in XML to develop the necessary templates for the corresponding plans in multi agent framework. The transformation rules are given below:

**R8: <EX-PATH> to <plan> rule:** The <EX-PATH> tag represents an execution path of an element in one scenario that is captured from a sequence diagram for a self-* property. This tag consists of one <TRIGGER> sub-tag and one or more <MESSAGE> sub-tags sent and received by different participating agents in runtime. The <TRIGGER> tag determines the triggering event of the execution path and consists of a sender, an event and a receiver. For reactive execution paths, the sender of the trigger event is the *environment*. Otherwise, for the proactive execution paths, the sender of the trigger

would be an internal agent. Similarly, the <MESSAGE> tags have a sender, an event and a receiver and specify the subsequent message activity of the agent. For each <EX-PATH> tag a <plan> tag will be created in the MAS model. This <plan> tag is in fact the header of the plan to be executed. The name of this <plan> tag will be the *name* attribute of the <EX-PATH> tag and its trigger will be the <EVENT> sub-tag's *name* attribute of the <TRIGGER> tag.

**R9: <MESSAGE> to <plan> rule:** Each <MESSAGE> tag is a triple consisting of <SENDER> tag specifying sender RAO, <EVENT> tag specifying the event being sent and <RECEIVER> tag specifying the receiver RAO. Each triple represents one action of the sequence diagram. For example if *CU1* sends *restart* message to *Sensor1* the following tags will capture this concept:

```
<MESSAGE>
  <SENDER name = "CU1"/>
  <EVENT name = "restart" type= "sync"/>
  <RECEIVER name = "Sensor1"/>
</MESSAGE>
```

This action can be mapped only to plans in agent model because the plans determine the behavior of agents. Each of these <MESSAGE> tags will be transformed to the corresponding JAVA code in the plan body of the agent to create the templates of the dynamic model. As an example, the previous <MESSAGE> tag will be transformed to the following JAVA code in the plan body:

```
public void body {
  ...
  IMessageEvent restart = CreateMessage("restart")
  try {
    SendMessageAndWait(restart, timeout)
    ...
  } catch (TimeoutException te) {
    ...
  }
  ...
}
```

In this example an instance of the *restart* message event is created in the plan body and then the plan sends this message and waits for the reply for a limited time period. This sending instruction is surrounded by a try-catch statement to catch the possible timeout exception. If the plan receives its response in time, it resumes the execution from the next statement right after the *SendMessageAndWait* command. If the receiving agent does not reply within the specified time period, the plan will fall into the catch block.

This example, although simple, illustrated an idea of the transformation process. However, the whole process is a little more complex. To better understand the rule, we will define three different types of <EVENT> sub-tag in <MESSAGE> tag:

- **Asynchronous message events:** These are the message events that are sent and the sender does not wait for any response from the receiver agent. In this case the value of type attribute of <EVENT> tag is *async*.

- **Synchronous message event without timeout:** This type of message event is sent and the sender waits for the response from the receiver until it gets the reply. The *type* attribute of the <EVENT> tag for this kind of message event is *sync*.

- **Synchronous message event with timeout:** This type of message event is similar to the previous one but with a limited time constraint. The timeout value is specified in the <TIMEOUT> sub-tag of <EVENT> tag with a minimum and maximum value.

- **Empty message event:** There is another type of message event that is the *empty* message. This type is used to capture the cases that the receiver does not respond in the determined time interval. For example if the *CU1* sends a *restart* message to

*Sensor1* and *Sensor1* does not respond to it and *CU1* must wait for the response minimum 10 time units and maximum 30 time units, the following <MESSAGE> tag will represent the situation:

```
<MESSAGE>
<SENDER name = "Sensor1"/>
<EVENT type= "sync">
      <TIMEOUT min = 10 max = 30/>
</EVENT>
<RECEIVER name = "CU1"/>
</MESSAGE>
```

For each of the message events there is a solution to transform it to the MAS meta-model. In general all of the event types are transformed to JAVA code that sends a message event. The only difference is how the message is sent and how the plan deals with the response to the message.

**R10: Asynchronous message event rule:** The simplest one of the four message types, asynchronous message event with the following format:

```
<MESSAGE>
  <SENDER name = "sender_rao"/>
  <EVENT name = "message_name" type= "async"/>
  <RECEIVER name = "receiver_rao"/>
</MESSAGE>
```

Will be transformed to the following JAVA code in the agents plan body:

```
public void body {
  ...
  IMessageEvent async_message = CreateMessage("message_name");
  try {
    SendMessage(async_message);
    ...
  } catch (Exception e) {
    ...
  }
  ...
}
```

This program code consists of an instruction to instantiate a message event of class IMessageEvent and the SendMessage() method to send the message event. The SendMessage() method is enclosed in a try-catch statement to catch any possible exceptions. All the statements will be surrounded in the body method of the plan.

**R11: Synchronous message event without timeout rule:** This type of message event will be sent by the sender agent and it will wait for a reply from the target agent. There is no waiting time limitation for this message event and the plan will be suspended until it gets the desired response from the destination agent. The following definition is a typical definition of this type of message event:

```
<MESSAGE>
  <SENDER name = "sender_rao"/>
  <EVENT name = "message_name" type= "sync"/>
  <RECEIVER name = "receiver_rao"/>
</MESSAGE>
```

The transformation of the example is the program code in JAVA in the plan body that is created as follows:

```
public void body {
  ...
  IMessageEvent sync_message = CreateMessage("message_name");
  try {
    IMessageEvent reply_message = SendMessageAndWait(sync_message);
    ...
  } catch (Exception e) {
    ...
  }
  ...
}
```

According to the example the transformation process will instantiate the corresponding message event class using CreateMessage() method. It will then send the created message instance by SendMessageAndWait() method and will wait for the reply to assign it to the

reply_message message event. This method is inside a try-catch statement to handle different exceptions. If the plan sends the message successfully and receives the reply as expected, it will continue its execution right after the SendMessageAndWait() sentence. If during this process any exception occurs, the plan will execute the catch block.

**R12: Synchronous message event with timeout rule:** This message event is exactly the same but with a time limitation. This timeout interval is specified in the <TIMEOUT> sub-tag of the <EVENT> tag. The attribute *min* of <TIMEOUT> tag defines the minimum timeout value and the attribute *max* determines the maximum timeout value:

```
<MESSAGE>
  <SENDER name = "sender_rao"/>
  <EVENT name = "message_name" type= "sync">
    <TIMEOUT min = interger max = integer/>
  </EVENT>
  <RECEIVER name = "receiver_rao"/>
</MESSAGE>
```

In the JAVA code that is the result of the transformation process, the plan will create an instance of the corresponding message event. The SendMessageAndWait() method will be used to send the message and wait for a specified time period. Since the time period is declared as a minimum and a maximum value, there will be a waitFor() method before sending the message to make sure that the plan will wait for the minimum time units specified. The plan will use the try-catch statement to be sure to catch the timeout exception.

```
public void body {
  ...
  IMessageEvent sync_message = CreateMessage("message_name");
  try {
    waitFor(min);
    IMessageEvent reply_message = SendMessageAndWait(sync_message, max - min);
    ...
  } catch (TimoutException te) {  ...  }
  ...
}
```

**R13: Empty message event rule:** When an agent is waiting for a reply from another agent (whether by any of waitFor() methods or by getInitialEvent() method), it is possible that it does not receive any response and it must react to this event. This is defined in RAS model by empty messages and will be transformed to the action being included in the catch block of the timeout exception. For instance if *CU1* sends a *restart* message to *Sensor1* and it sends no message in the specified time limit (minimum 10 and maximum 30 time units) to *CU1*, this will be captured in RAS as the following sequence:

```
<MESSAGE>
  <SENDER name = "CU1"/>
  <EVENT name = "restart" type= "sync"/>
  <RECEIVER name = "Sensor1"/>
</MESSAGE>
<MESSAGE>
  <SENDER name = "Sensor1"/>
  <EVENT type= "sync">
    <TIMEOUT min = 10 max = 30/>
  </EVENT>
  <RECEIVER name = "CU1"/>
</MESSAGE>
```

The transformation process will create the following JAVA code for the two <MESSAGE> tags in the example:

```
public void body {
  ...
  IMessageEvent restart = CreateMessage("restart");
  try {
    waitFor(10);
    IMessageEvent heartbeat = SendMessageAndWait(restart, 20);
    ...
  } catch (TimoutException te) {
    //The code to send a request to CU8 to search for a Sensor agent will go here.
    ...
  }
  ...
}
```

In this code an instance of *restart* message is instantiated. The process continues by waiting 10 units of time as specified in the empty message event. Then the agent will

send the message and will wait for 20 time units that is the difference between max and min values in <TIMEOUT> sub-tag. In the case of no response (empty message) the plan will fall in the catch block and the agent can send a message to *CU8* agent to request another *Sensor* agent to substitute *Sensor1*.

**R14: <REACTIVE-BEH> vs. <SELF-PROP>:** There are two principal behavioral tags that will contain <EX-PATH> tags; the <REACTIVE-BEH> tag and the <SELF-PROP> tag. The reactive behavior of the element in RAS model is defined by <REACTIVE-BEH> tag. The execution paths under this tag are triggered with an external event from environment or another agent.

On the other hand the proactive behavior of the element in RAS model is determined by <SELF-PROP> tag. The difference between this type of behavior and the reactive behavior of the element is the way the execution path is triggered. In this case the trigger is an internal event that is in fact the result of internal status of elements in communication. For example if a *RAO* decides to trigger an execution path to adapt itself to a situation, that will be considered as proactive behavior. The proactive behavior of elements in RAS model can be mapped to the goal definition of multi agent systems. From the four principal goal types in MAS, the <achievegoal> tag will be a good match for <SELF-PROP> tag.

**R15: <GOAL> to <achievegoal> rule:** The <GOAL> sub-tag under <SELF-PROP> tag represents the proactive behavior of RAS model and will be transformed to <achievegoal> tag in MAS model. The *name* attribute of <achievegoal> tag will be the *name* attribute of <GOAL> tag. The plan of the agent that corresponds to the <EX-

PATH> tag of the RAS element will have a trigger other that event messages. In fact the plan will be triggered by the goal. This goal will be created and dispatched in the plan that currently executes and evaluates the internal status of the agent (the monitoring plan of the RAO). In fact to implement this rule, there are three principal components that must be taken into account:

*1- Monitoring plan*

For the agents that represent RAOL elements, a plan will be created to monitor the other RAO elements in the group. In this plan the agent regularly sends messages to query the status of group agents. This plan will be started when the agent is activated. To specify this part we need a configuration section in RAC definition.

```
public void body {
  ...
  while (agentActive) {
    IMessageEvent checkstatus = CreateMessage(“checkstatus”);
    try {
      waitFor(10);
      IMessageEvent heartbeat = SendMessageAndWait(checkstatus, 20);
      agentActive = true;
    } catch (TimoutException te) {
      //this part creates and dispatches self tolerance goal.
      IGoal sf = createGoal(“self_tolerance”);
      dispatchTopLevelGoal(rs);
      agentActive = false;
      ...
    }
    ...
  }
}
```

*2- Self-* property goal*

A <achievegoal> will be created having the same name as the <GOAL> tag in the RAS specification. This goal will be created under the <goals> tag in the ADF file of the agent representing RAOL element.

*3- Plan of the goal*

The plan header for the execution path will have a <trigger> tag specifying the goal that triggers it. If the *name* attribute of the goal is *goal_name* the plan header will be like the following definition in ADF file:

```
<plan name = "self_tolerance">
  <body class = "selfTolerancePlan"/>
    <trigger>
      <goal ref = "goal_name"/>
    </trigger>
</plan>
```

## 3.6. Example

The following example will illustrate the transformation process using a simple model. In this example there are two RACs that communicate with each other. RAC1 is named Production Robot1 that consists of three RAOs including Sensor1, Drill1 and CU1 as the RAOL of the RAC. The second RAC is called Production Robot8 also having three RAOs consisting of Sensor8, Drill8 and CU8 as its RAOL. These two RACs can communicate with one another via their RAOLs as the control unit. Also the three RAOs inside the RACs are capable of interacting with each other.

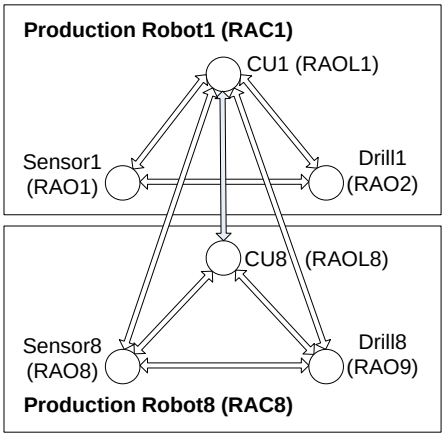


**Figure 3.8:** Two RACs communicating with each other.

Figure 3.8 represents the static model of these two RACs. In this Figure, CU1 can communicate with CU8 as well as Sensor8 and Drill8. The static view of the model can be serialized into XML format. Figures 3.9, 3.10 and 3.11 show the captured XML representation of RAO1, RAO2 and RAOL1 and Figures 3.12 and 3.13 show the definition of RAC1 and RAC8 again in XML format.

```
<RAO name = "Sensor1">
   <REACTIVE-ATOMIC>
      <TRIGGER name= "restart"/>
      <PLAN name= "restart-plan"/>
      <RESPONSE name= "heartbeat"/>
   </REACTIVE-ATOMIC>
</RAO>
```

**Figure 3.9:** The XML definition of RAO1.

```
<RAO name = "Sensor8">
   <REACTIVE-ATOMIC>
      <TRIGGER name= "request"/>
      <PLAN name= "request-plan"/>
      <RESPONSE name= "confirmed"/>
   </REACTIVE-ATOMIC>
   <REACTIVE-ATOMIC>
      <TRIGGER name= "register"/>
      <PLAN name= "register-plan"/>
      <RESPONSE name= "heartbeat"/>
   </REACTIVE-ATOMIC>
   <REACTIVE-ATOMIC>
      <TRIGGER name= "deregister"/>
      <PLAN name= "deregister-plan"/>
      <RESPONSE name= "confirmed"/>
   </REACTIVE-ATOMIC>
</RAO>
```

**Figure 3.10:** The XML definition of RAO8.

```
<RAO name = "CU1">
   <REACTIVE-ATOMIC>
      <TRIGGER name= "sensor-found"/>
      <PLAN name= "sensor-found-plan"/>
      <RESPONSE name= "register"/>
   </REACTIVE-ATOMIC>
</RAO>
```

**Figure 3.11:** The XML definition of RAOL1 (CU1).

In the previous three figures, the XML definition of atomic behaviors in Sensor1, Sensor8 and CU1 is specified. For each atomic behavior a <REACTIVE-ATOMIC> tag is specified. There are three sub-tags, <TRIGGER>, <PLAN> and <RESPONSE>, that define each atomic behavior. Each atomic behavior is a reactive behavior that is triggered by the message event specified in *name* attribute of <TRIGGER> sub-tag. This trigger causes the RAO to respond to the trigger by executing a plan specified in the *name* attribute of <PLAN> sub-tag. This plan prepares and sends a response message event specified in the *name* attribute of <RESPONSE> sub-tag. For example in Figure 3.9 when Sensor1 receives a *restart* message event it activates its *restart-plan* plan. In this plan the RAO element creates and sends a *heartbeat* message event in response to the trigger.

```
<RAC name = "Production Robot 1">
   <MEMBERS>
      <MEMBER name = "CU1"/>
      <MEMBER name = "Sensor1"/>
      <MEMBER name = "Drill1"/>
   </MEMBERS>
   <INTERACTIONS>
      <INTERACTION source = "CU1" name = "restart" target = "Sensor1"/>
      <INTERACTION source = "Sensor1" name = "heartbeat" target = "CU1"/>
      <INTERACTION source = "CU1" name = "request_sensor" target = "CU8"/>
      <INTERACTION source = "CU1" name = "register" target = "Sensor8"/>
      <INTERACTION source = "CU1" name = "take_over_sensor" target = "Drill1"/>
      <INTERACTION source = "Drill1" name = "confirmed" target = "CU1"/>
```

```
        <INTERACTION source = "CU1" name = "take_over_sensor" target = "Drill8"/>
        <INTERACTION source = "Drill8" name = "confirmed" target = "CU1"/>
    </INTERACTIONS>
    <LEADER name = "CU1"/>
    <REPOSITORY>
        <PROPERTY name="timeout" type="String">millisecond</PROPERTY>
    </ REPOSITORY>
</RAC>
```

**Figure 3.12:** The XML definition of the static view of RAC1.

```
<RAC name = "Production Robot 8">
    <MEMBERS>
        <MEMBER name = "CU8"/>
        <MEMBER name = "Sensor8"/>
        <MEMBER name = "Drill8"/>
    </MEMBERS>
    <INTERACTIONS>
        <INTERACTION source = "CU8" name = "restart" target = "Sensor8"/>
        <INTERACTION source = "Sensor8" name = "heartbeat" target = "CU8"/>
        <INTERACTION source = "CU8" name = "sensor_found" target = "CU1"/>
        <INTERACTION source = "Sensor8" name = "heartbeat" target = "CU1"/>
    </INTERACTIONS>
    <LEADER name = "CU1"/>
    <REPOSITORY>
        <PROPERTY name="timeout" type="String">millisecond</PROPERTY>
    </ REPOSITORY>
</RAC>
```

**Figure 3.13:** The XML definition of the static view of RAC8.

As shown in Figure 3.12, in the RAC definition the <MEMBERS> tag specifies the RAOs belonging to the RAC. This includes CU1, Sensor1 and Drill1 for Production Robot1 and CU8, Sensor8 and Drill8 for Production Robot8. The <INTERACTIONS> tag specifies the communication between different RAOs of the RAC and the name of the events assigned to each interaction. For example the first <INTERACTION> tag in Production Robot1 denotes that there is an event message from CU1 to Sensor1 called *restart*. The communication between two RACs, i.e., (CU1, CU8), (CU1, Sensor8), (CU1, Drill8), must be specified in RACG definition. To make the example simpler, we have

not defined the RACG specifications and we have considered that RAC1 has access to the knowledge of RAC8 components. Otherwise, the knowledge of interactions between the RAC components must be defined in RACG.

The behavior of RAS is specified using sequence diagrams. For instance the fault tolerance property of the architecture specified in Figure 3.8 is shown in the sequence diagram given in Figure 3.14 [19].

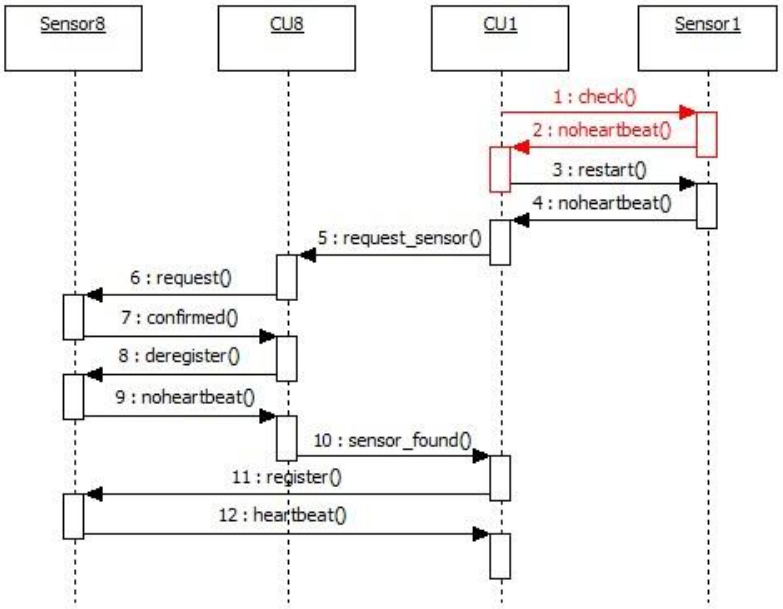


**Figure 3.14:** The sequence diagram representing the fault tolerance property of RAC1.

Figure 3.14 shows an execution path triggered by the CU1 element to achieve a goal that fulfills the fault tolerance property. In this sequence diagram, CU1 checks the status of Sensor1 by sending a *check* message. If Sensor1 does not reply in a limited time period, it is denoted here by a *noheartbeat* message. This means that if CU1 does not get a response in this time period it will trigger its fault tolerance plan that starts from the 3rd

step. In this case CU1 sends a *restart* message to Sensor1. If Sensor1 does not reply again, CU1 will request a sensor from CU8 by sending *request_sensor* message to it.

```
<RAC name = "Production Robot 1">
  <SELF-PROP>
    <GOAL name = "self-tolerance" path = "sensor-recovery"/>
    <EX-PATH name = "sensor-recovery">
      <TRIGGER>
        <SENDER name = "CU1"/>
        <EVENT name = "check"/>
        <RECEIVER name = "Sensor1"/>
      </TRIGGER>
      <MESSAGE>
        <SENDER name = "CU1"/>
        <EVENT name = "restart" type= "event-type">
          <TIMEOUT min= 10 max = 30/>
        </EVENT>
        <RECEIVER name = "Sensor1"/>
      </MESSAGE>
      <MESSAGE>
        <SENDER name = "CU1"/>
        <EVENT name = "request_sensor" type= "event-type"/>
        <RECEIVER name = "CU8"/>
      </MESSAGE>
      <MESSAGE>
        <SENDER name = "CU8"/>
        <EVENT name = "sensor_found" type= "event-type"/>
        <RECEIVER name = "CU1"/>
      </MESSAGE>
      <MESSAGE>
        <SENDER name = "CU1"/>
        <EVENT name = "register" type= "event-type"/>
        <RECEIVER name = "Sensor8"/>
      </MESSAGE>
      <MESSAGE>
        <SENDER name = "Sensor8"/>
        <EVENT name = "heartbeat" type= "event-type"/>
        <RECEIVER name = "CU1"/>
      </MESSAGE>
    </EX-PATH>
  </SELF-PROP>
</RAC>
```

**Figure 3.15:** The XML definition of the behavioral fault tolerance property of RAC1.

```
<RAC name = "Production Robot 8">
   <SELF-PROP>
      <GOAL name = "self-tolerance-reply" path = "sensor-supply"/>
      <EX-PATH name = "sensor-supply">
         <TRIGGER>
            <SENDER name = "CU1"/>
            <EVENT name = "request_sensor"/>
            <RECEIVER name = "CU8"/>
         </TRIGGER>
         <MESSAGE>
            <SENDER name = "CU8"/>
            <EVENT name = "request" type= "event-type"/>
            <RECEIVER name = "Sensor8"/>
         </MESSAGE>
         <MESSAGE>
            <SENDER name = "Sensor8"/>
            <EVENT name = "confirmed" type= "event-type"/>
            <RECEIVER name = "CU8"/>
         </MESSAGE>
         <MESSAGE>
            <SENDER name = "CU8"/>
            <EVENT name = "deregister" type= "event-type"/>
            <RECEIVER name = "Sensor8"/>
         </MESSAGE>
         <MESSAGE>
            <SENDER name = "CU8"/>
            <EVENT name = "sensor_found" type= "event-type"/>
            <RECEIVER name = "CU1"/>
         </MESSAGE>
      </EX-PATH>
   </SELF-PROP>
</RAC>
```

**Figure 3.16:** The XML definition of the behavioral fault tolerance property of RAC8.

CU8 when receiving this message searches its lookup directory to find the sensor and sends a *request* message to see if it is available. By receiving a *confirmed* message from the sensor, CU8 will send a *deregister* message to Seosor8. After receiving the deregistration confirmation, CU8 will send a *sensor_found* message to CU1 including the address of Sensor8. Finally, CU1 will register Sensor8 to its directory by sending a

*register* message to it and Sensor8 will send a *heartbeat* message to CU1 in response. This execution path will be captured in XML format as shown in Figures 3.15 and 3.16.

The static architecture of two RACs shown in previous Figures, together with the dynamic behavior of these two RACs and the RAOs inside them represented in XML format will be served as the input to the transformation process. As discussed earlier, the output of this process will include the ADF files denoting the agents in XML format and the corresponding JAVA programs for the plans.

In the first step of transformation process, according to the rule **R3**, for each RAO definition in XML format an agent will be created. Figure 3.17 shows the elements that will be created in MAS model for this example. Each agent is an XML ADF file that conforms to the Jadex specifications.

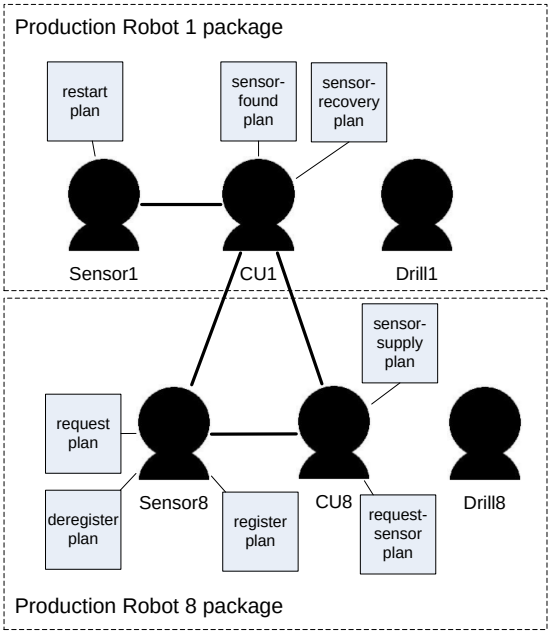


**Figure 3.17:** The MAS model created from model transformation process.

For example the CU1.xml file that represents a RAOL definition in RAS model, will be transformed to CU1.agent.xml agent definition file that contains the Jadex format agent definition tags. There are some specific header tags and tags to import Jadex libraries used in the agent that must be created in the ADF of the agent in this step (Please, see the Appendix for more details.):

CU1.xml -> CU1.agent.xml

Sensor1.xml -> Sensor1.agent.xml

In the second step, the RAC definition will be used to group the agents in a package to show that they belong to the same RAC. The rules that are applied in this step are **R1** and **R2** rules, which group the agents in a package that belong to one RAC:

CU1, Sensor1, Drill1 -> Production Robot 1 package

CU8, Sensor8, Drill8 -> Production Robot 8 package

In the third step, according to the rule **R4**, the <INTERACTIONS> tag will be utilized to create the event messages between different RAOs. Using this tag, the transformation process will create two message events, one message of type “output” in the source RAO having the name specified in the *name* attribute and one message of type “input” in the target RAO having the same name.

Interaction tag -> “output” message in source RAO, “input” message in target RAO

For example, the following <INTERACTION> tag of the input file:

```
<INTERACTION source="CU1" name="restart" target="Sensor1"/>
```

will be transformed to the following <messageevent> tag in *CU1.agent.xml* output file:

```
<messageevent name="restart" type="fipa" direction="send">
  <parameter direction="fixed" class="String" name="performative">
    <value>SFipa.REQUEST</value>
  </parameter>
</messageevent>
```

and to the following <messagevent> tag in *Sensor1.agent.xml* output file:

```
<messageevent name="restart" type="fipa" direction="receive">
  <parameter direction="fixed" class="String" name="performative">
    <value>SFipa.REQUEST</value>
  </parameter>
</messageevent>
```

In the fourth step, the process will use the <REPOSITORY> tag to create the beliefs in RAOL agents. For each <PROPERTY> tag in the repository a belief will be created in the ADF file with the same type and for each value of this property a fact will be added to this belief. This step refers to the rule **R6**.

Property in Repository -> belief

For example the following <PROPERTY> tag:

```
<PROPERTY name="timeout" type="String">milllisecond</PROPERTY>
```

Will create the following <belief> tag in CU1.agent.xml ADF file for CU1 RAOL:

```
<belief name="timeout" class="string">
  <fact>millisecond</fact>
</belief>
```

In the fifth step, according to rule **R9**, the transformation process will take advantage of atomic behaviors defined inside RAO XML files to create atomic plans for agents. In this example, the atomic behaviors consist of *restart*, *request*, *register* and *deregister*. The transformation process will generate one Java file (for example *restart.java*) for each of these atomic behaviors. These Java programs contain a *body()* method that is executed when the plan is triggered.

restart -> restart plan

request -> request plan

These Java programs are code templates that only contain the necessary code to fulfill the fault-tolerance property. They must be customized by the programmers according to the capabilities of each multi-agent system.

In the last step, using self-* property definitions in RAC, the transformation process will create fault-tolerance plans for CU2 agent and its complementary behavior in CU8 agent. The execution path in self property tag will determine the messages sent by the RAOL and actions done in response. According to the rules **R8**, **R9** and **R14** discussed before in this chapter, each of the <MESSAGE> tags in the execution path will create the corresponding Java program statements in the plan.

sensor-recovery -> sensor-recovery plan

sensor-supply -> sensor-supply plan

For example, the tag <EX-PATH name = “sensor-recovery”> will generate a plan *sensor-recovery.java* with a *body()* method that contains message sending and receiving commands according to the input execution path and the rules **R11**, **R12** and **R13**.

## 3.7. Conclusion

In this chapter, we discussed the transformation approach to convert the RAS model to MAS model. The input model to this model transformation method is captured in XML format from the RAS architecture. The output model is the agent definition templates in Jadex including the Agent Definition File (ADF) in XML format as well as plan source codes in Java. The transformation rules provide the conversion of input model to output

model. These transformation rules are implemented in XSLT [45], a model transformation tool of XML-based models and its language called XPath to create Jadex templates from RAS XML definition.

The input model or LHS discussed in this chapter is created using a grammar based on the Extended BNF standard [24]. This grammar is used to create the static and the dynamic model of RAS in XML format to permit the easy transformation of the LHS. The output model or RHS is the Jadex BDI model that is a powerful Java based agent programming environment. The objective of this chapter was to represent the important features and specifications of this model to have a better view of the output format. The chapter presented the transformation rules according to the input and output model discussed before. These transformation rules take each input item and transform it to the corresponding element in the output file. Finally, the chapter discussed a simple example of transformation process to better illustrate the different concepts.

# Chapter 4: Implementation of Fault-Tolerance in Case Study

In this thesis, we present the implementation of fault-tolerance mechanism in the Marsworld case study using Jadex, a BDI-based multi-agent programming add-in with Eclipse Java development environment.

In Chapter 3, we proposed a model transformation approach to create agent templates of MAS in Jadex from RAS. These agent templates consist of ADF files in XML format for agent definitions and Java programs for agent plans. In this chapter, we customize these agent templates for Marsworld case study to implement it in an executable example for the purpose of observing its correctness.

## 4.1. The Marsworld Case Study as a MAS

To illustrate how the substitutability property (see Chapter 2) guarantees the fault-tolerance property of the RAS meta-model, the Marsworld case study has been used and implemented in Jadex. To reduce the complexity of the case study, we have modeled the RAC level as the lowest layer. This means that in this case study we will map the RAC (not the RAO) in the RAS meta-model to the *agent* in the MAS meta-model. In the Marsworld case study there are five types of agents (robots), *Manager*, *Supervisor*, *Sentry*, *Production*, and *Carry* agents.

*Manager* creates and manages the *Supervisor* agents. This agent is the starting point of the system and can interact with the user. In fact the user interface of the system is implemented in this agent to provide a tool for human interaction. Figure 4.1 depicts the main components of the *Manager* agent in Jadex. As shown in this Figure, there are two principal plans in this agent, *MarsworldGUI* and *StarterPlan*. The *MarsworldGUI* plan is responsible for initializing and handling the graphical part of the case study as well as the interactions of the user with the program. This includes the mouse click events that the user performs to kill an agent. In this case, the Manager agent sends a *request_shutdown* message to the agent that has received the click event. The *StarterPlan* plan initiates a *supervisor* agent and assigns it to an exploration area. The white shapes in Figure 4.1 are from the Marsworld example [5, 13].

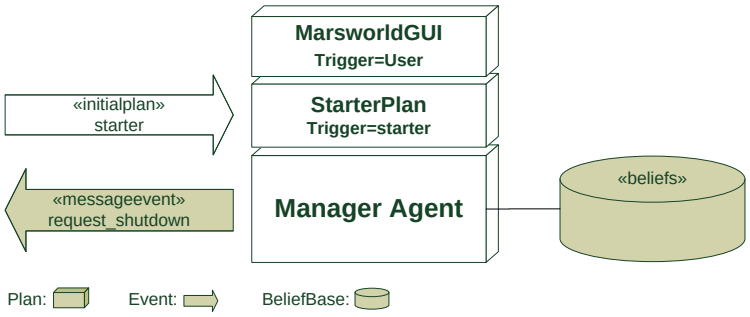


**Figure 4.1:** The Jadex architecture of the *Manager* agent.

*Supervisor* is in charge of an exploration group to exploit ore mines. After its creation, this agent initiates a number of *Sentry* agents to find and analyze the ore targets in the exploration area assigned by *Manager*. Besides, the *Supervisor* agent has the ability to search and find target mines. If a target is found, this agent assigns the task of analyzing the target to an available *Sentry* agent and subsequently forms a group of

*Production* and *Carry* agents to do the exploiting task. As presented in Figure 4.2, the *Supervisor* agent has a recovery plan for each existing agent type of the exploration group. These recovery plans consist of *RecoverCarryPlan*, *RecoverProductionPlan* and *RecoverSentryPlan*. The Supervisor agent has another plan called *CheckAgentsPlan* which checks regularly the group member agents and creates a recovery goal for any agent that is damaged. Subsequently, the created goal triggers the corresponding recovery plan. The *Supervisor* agent activates the recovered agent by sending the appropriate request message to it such as *request_producer* for the *Production* agent.

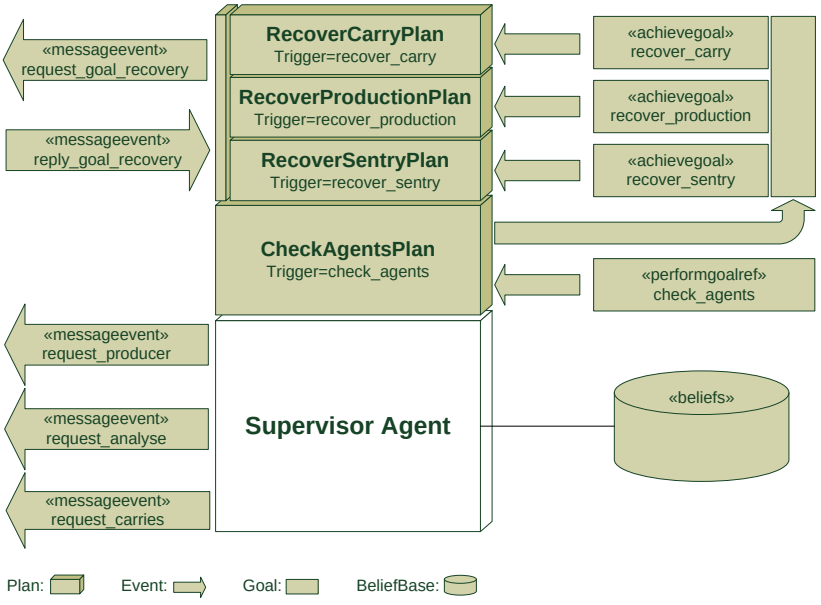


**Figure 4.2:** The Jadex architecture of the *Supervisor* agent.

The *Sentry* agent analyzes the mines that it has found or are assigned to it by the *Supervisor* agent. After finishing the analyzing process, it calls the available *Production* agents to exploit ore in the target mine. Figure 4.3 depicts the Jadex architecture of the *Sentry* agent. The recovery plans and their triggering messages and goals are presented in

this Figure. The *RecoverPlan* plan is triggered by reception of *request_goal_recovery* message and subsequently creates *recover_goal* goal that triggers the *RecoverGoalPlan* plan. The location recovery is accomplished by *RecoverLocationPlan* plan triggered by *request_location_recovery* message. To simulate the damage of the *Sentry* agent for fault-tolerance tests, a plan called *ShutdownPlan* is provided in this agent that is triggered by the user among the *Manager* agent that sends a *request_shutdown* message. The white shapes are from the Marsworld example [5, 13].

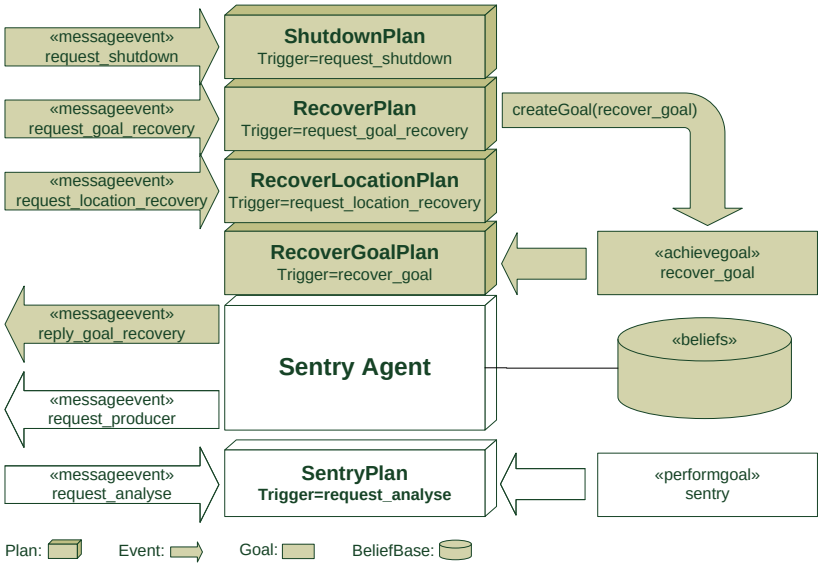


**Figure 4.3:** The Jadex architecture of the *Sentry* agent.

*Production* is called by *Sentry* agent to exploit ore in a specific target mine. Figure 4.4 shows the principal components of the *Production* agent. Similar to the *Sentry* agent, there are three plans to accomplish the recovery process. Besides, the *shutdownPlan* plan is provided to simulate an unexpected accident that lead to the crash of the agent. In Figure 4.4, the white shapes refer to the components that come from the Marsworld example [5, 13]. After finishing the production task, the *Production* agent calls the available *Carry* agents to carry produced ore in the mine to the home base.

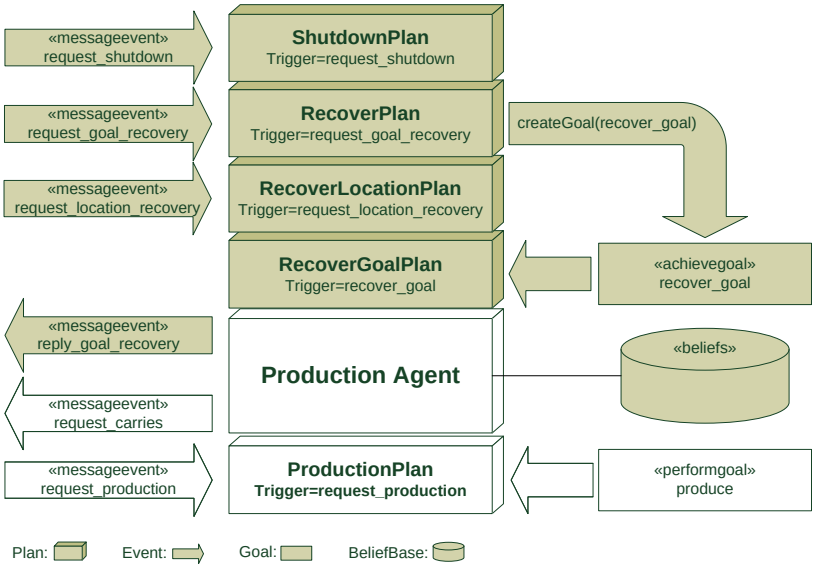


**Figure 4.4:** The Jadex architecture of the *Production* agent.

The *Carry* agent has a limited capacity of ore so that it travels between the target mine and home base [5]. From fault-tolerance point of view, the *Carry* agent is quite similar to the *Production* agent. Figure 4.5 represents the most important components of the *Carry* agent. The white shapes refer to the Marsworld example [5, 13].

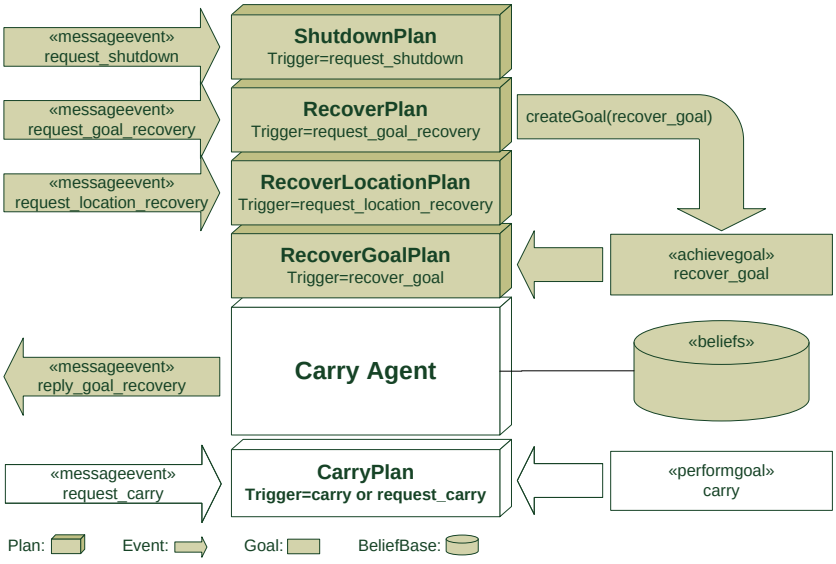


**Figure 4.5:** The Jadex architecture of the *Carry* agent.

## 4.2. Fault-Tolerance in Marsworld

In order to simulate the malfunctioning of one agent in terms of fault-tolerance property verification, the user's click on the agent in GUI is considered as a signal to disable it. This is done inside the mouse click event listener of the environment panel in the *MarsworldGUI* plan of the manager agent. If the x and y of the clicked point falls inside the surface of any agent, it creates a message event that tells the agent to shutdown itself. Figure 4.6 depicts the shutdown sequence diagram.

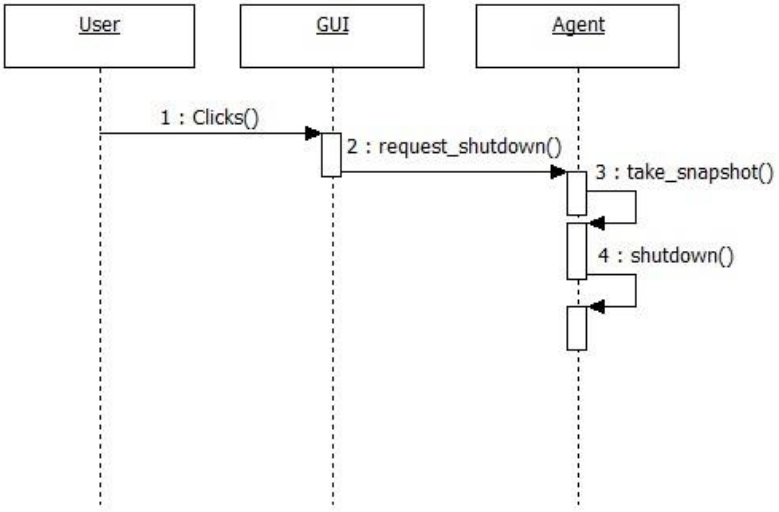


**Figure 4.6:** The Shutdown sequence diagram.

For each agent there is a *shutdown* plan that takes a snapshot of the agent and pushes it into a queue and then shuts down the agent. This snapshot is retrieved later by the Supervisor agent to recover the damaged agent and consists of: 1) the agent snapshot: last updated copy of the agent's belief-base; 2) goal snapshot: information about the current goal of the agent; and 3) message snapshot: the message event queue representing the

information of messages that the agent has received. Figure 4.7 illustrates the pseudo code of the *ShutdownPlan* plan for each group agent.

```
begin
   create new agent snapshot
   copy dynamic belief-base information to agent snapshot
   copy dispatched goals to agent snapshot
   copy event message queue to agent snapshot
   add the agent snapshot to agent snapshot queue
   kill agent
end
```

**Figure 4.7:** The *ShutdownPlan* pseudo-code of the group agents.

The *Supervisor* agent has a perform goal named *check_agents* (Figure 4.8) that checks continuously the state of the agents belonging to its exploration group by triggering the plan *CheckAgentsPlan*.

```
begin
   get online information of agents from repository
   for (each agent in the group) {
      get the status of the agent
      if (the agent is inactive) {
         get the agent type
         if (the agent is a carry_agent) {
            create goal recover_carry
            dispatch top level goal recover_carry
         }
         if (the agent is a production_agent) {
            create goal recover_production
            dispatch top level goal recover_production
         }
         if (the agent is a sentry_agent) {
            create goal recover_sentry
            dispatch top level goal recover_sentry
         }
      }
   }
end
```

**Figure 4.8:** The *CheckAgentsPlan* plan of the *Supervisor* agent.

This plan, which is a Java program stored in `CheckAgentsPlan.java` file, has a method called *body()*. In the *body()* method, which is executed when the plan is triggered

by the corresponding goal, if the *Supervisor* agent detects any inactive agent in the group, it verifies its type and then selects the appropriate recovery plan for that type of agent and subsequently creates the respective top level goal for its recovery. Figure 4.8 depicts the pseudo code of *CheckAgentsPlan* plan of the *Supervisor* agent.

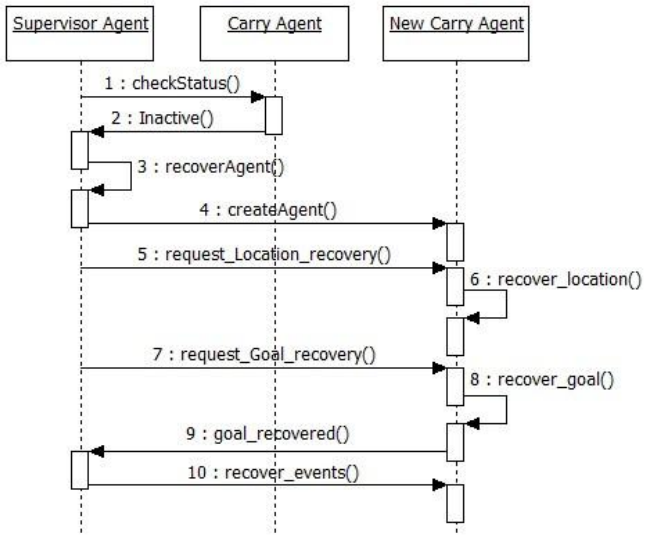


**Figure 4.9:** The Carry recovery sequence diagram.

The *Supervisor* agent has a recovery plan for each of the group agents including the *Sentry* agent, the *Production* agent, and the *Carry* agent. For example if the *Carry* agent is damaged, the *Supervisor* selects the *RecoverCarryPlan* to recover the *Carry* agent. Figure 4.9 illustrates the sequence diagram of recovery process for the *Carry* agent.

The *RecoverCarryPlan* plan, which is also a Java program having a *body()* method, consists of four steps to recover the *Carry* agent: 1) it creates a new *Carry* agent from scratch; 2) it recovers the miscellaneous agent information or in fact the belief-base such as the location of the agent; 3) it deals with the goal recovery; and 4) it recovers the

message event queue of the *Carry* agent. Figure 4.10 shows the pseudo code of the *RecoverCarryPlan* plan of the *Supervisor* agent.

```
begin
   create goal ams_create_agent
   dispatch sub goal ams_create_agent and wait
   create message event request_location_recovery
   set the content of message event to location of agent snapshot
   send message request_location_recovery and wait
   create message event request_goal_recovery
   get the goal snapshot from agent snapshot
   set the content of message event to goal snapshot
   send message request_goal_recovery and wait
   get message event queue
   while (there is a message event in the queue) {
      create message event request_carries
      set the receiver of the message as the agent identifier
      set the content of the message to the content of the message event from queue
      send message request_carries
   }
end
```

**Figure 4.10:** The *RecoverCarryPlan* plan of the *Supervisor* agent.

## 4.3. Agent Creation

To create a new agent the *ams_create_agent* goal of *amscap* capability of *AMS* agent is used. Capabilities in Jadex are predefined libraries that provide ready-to-use functionalities for different purposes such as agent creation, search, shutdown, etc [13]. When an agent of a certain type is created, in fact the static initial state of that agent type is recovered automatically. This primary state consists of the initial values and conditions of beliefs, goals and plans. For example any agent has a belief called *my_vision* indicating its visual perception. This belief has different values for different types of agents. For instance, the default value of *0.05* for this belief in *Carry* agent denotes that this agent can only sense the objects in this range. When a new agent of type *Carry* is created, this belief is initiated to the default value *0.05*. This type of information can be

considered as static initial status of an agent and is recovered in agent creation phase. There is another type of status information that is dynamic and changing over time. For instance, the location of an agent is dynamic since it is changing when the agent is moving around.

The recovery of dynamic status of the agents is based on the status snapshots taken at shutdown moment of the agent. This information is a copy of the current status of beliefs, goals and events received by the agent that is captured and stored when the agent is shutdown. For example, the current location of the *Carry* agent is stored in an object of class *AgentSnapshot* that takes the current location of the agent in shutdown plan. In order to simulate the ongoing access of the *Supervisor* agent to the information of its group, there must be a way to inform it of the current status of its agents. In real world, this information is stored in log files in a safe place that is not damaged easily, such as the *black box* of an airplane. In our research the *AgentSnapshot* class contains this important dynamic information like location, the stack of goal snapshots and the name of the damaged agent. The name of the damaged agent is kept since we will use it to access the previous message event queue to recover it.

## 4.4. Recovery of Location using Agent Snapshot

The *Supervisor* agent polls the current agent snapshot from the agent snapshot queue and creates the message event *request_location_recovery* and sends this message to the newly created *Carry* agent to recover its location. When the location of the *Carry* agent is recovered, it starts its tasks from the recovered location. We have chosen the location to recover since in the Marsworld Jadex example it is more tangible and can be observed in

the scenario. In real world the location of a robot that is damaged may be of no interest to be recovered. Figure 4.11 shows the *RecoverLocationPlan* plan of the *Carry* agent.

```
begin
   while (not end of mission) {
      wait for message request_location_recovery
      get content of the request_location_recovery message as location
      set the fact of the belief my_location to location
      create goal walk_around
      dispatch to level goal walk_around
   }
end
```

**Figure 4.11:** The *RecoverLocationPlan* plan of the *Carry* agent.

When this message event is received by the new *Carry* agent, it triggers the *RecoverLocationPlan* plan, where, the agent is waiting for the *request_location_recovery* message and when receiving, it restores the location of the agent from agent snapshot and sets the current location of the agent to this value and then creates the *walk_around* goal to start the walking of the agent from this location. The *walk_around* goal is a *perform* goal that is followed by the agent when there is nothing else to do. On the other hand, when an agent is moving around it can find new sources of ore and inform the *Supervisor* agent of their existence. This *walk_around* goal is inhibited if in the next step of recovery the *carry_ore* goal is recovered because the latter has a higher priority to the former.

## 4.5. Current Goal in Hand

The current plan that the agent is pursuing must be recovered. For example, if a *Carry* agent has loaded ore and wants to deliver it to the home base, it is in the middle of the *carry_ore* plan. The only way to get a snapshot of the plan execution is to store useful variables from different steps of the plan (commit and rollback). For example, if the

loaded ore is zero, it means that the *Carry* agent wants to move to the target mine and reload ore; if it is greater than zero, it means that the *Carry* agent is moving from the target mine to the home base in terms of delivering its loaded ore.

To recover the current goal, the *Supervisor* agent takes advantage of the *GoalSnapshot* stack inside the *AgentSnapshot* class. The *Supervisor* agent creates a *request_goal_recovery* message event and puts the *GoalSnapshot* as its content and sends the request to the new *Carry* agent. If there is no goal to recover the value of *null* is set as the goal to recover. After sending the request, the *Supervisor* agent waits for the reply from the *Carry* agent to see if it has finished its recovery process. This is done by using the *sendMessageAndWait* method to establish a conversation between the two agents. The reason is that the new *Carry* agent has to finish the unfinished goal of the damaged agent before moving to its message event queue

```
begin
   get the goal snapshot from the log
   get the target mine location
   get the target mine ore amount
   get the ore load amount carried by the previous agent
   get the capacity of the agent
   while ((ore amount in the mine) or (ore load amount) is not zero) {
      if (ore load amount is zero) {
         create move_destination goal
         set the destination parameter to target mine
         dispatch subgoal move_destination to target mine and wait
         retrieve ore amount according to the capacity
      }
      if (ore load amount is more than zero) {
         create move_destination goal
         set the destination parameter to home base
         dispatch subgoal move_destination to home base and wait
         deliver ore amount loaded to the agent
      }
   }
end
```

**Figure 4.12:** The *RecoverGoalPlan* plan of the *Carry* agent.

to pick an event message to start a new *carry_ore* goal. In fact, by establishing a conversation between the two agents and waiting for the reply, the recovery plan in the *Supervisor* side is suspended until a response comes back from *Carry* agent. Figure 4.12 depicts the pseudo code for the *RecoverGoalPlan* plan of the *Carry* agent.

On receiving a request for the goal recovery, the *Carry* agent triggers its plan to recover the goal using the goal snapshot information, by which the *Carry* agent uses to identify the step of the task that the damaged agent was executing when a problem happened. In our example, the necessary data to recover the goal that is found in the goal snapshot object consists of:

- Goal type: identifying the type of the goal to be recovered, such as *carry_ore*.
- Target location: specifying the target location from which the *Carry* agent carries ore to the home base.
- Ore load: indicating the ore amount loaded to the *Carry* agent. This variable can be used as an indicator to determine whether the *Carry* agent is carrying ore to the home base or is moving to the target mine to reload ore.

In the *RecoverGoalPlan* plan the *Carry* agent restores the target location and ore load from the goal snapshot. If the ore load is zero, this means that the *Carry* agent has to move to the target mine to reload ore and carry it to the home base. Therefore, the starting point in this case will be moving to the target mine and reloading ore. If the ore load is greater than zero, this means that the damaged agent was carrying a certain amount of ore to the home base. In this case, the ore is loaded to the new *Carry* agent and then it moves to the home base to deliver ore. In fact the *RecoverGoalPlan* is a special copy of *CarryOrePlan* with a facility of conditional entrance points according to variable

checkpoints. This task continues until all the ore in that target is carried to the home base. After finishing the goal, the *Carry* agent pops the goal from the stack. The reason that the goal is not deleted from the queue in this point is that if anything happens to the new *Carry* agent in the middle of the recovery process, we will keep the recovery snapshot record in the stack for another new agent to recover it.

After finishing this task, the *Carry* agent creates its *carry* goal that listens to the *request_carry* message events. These message events can be from the *Supervisor* agent that is recovering the message event queue of the damaged agent or from the *Production* agents as expected in the normal behavior of the system. If this *carry* goal is not started, the *Carry* agent will not listen to *request_carry* event messages and these messages will not be captured.

In this point that the *Carry* agent is listening to *request_carry* message events, the *Supervisor* agent can start the recovery of the message events. Therefore, the *Carry* agent creates a reply message event named *reply_goal_recovery* in response to message *request_goal_recovery* of the *Supervisor* agent. This action activates again the recovery plan in the *Supervisor* side.

## 4.6. Message Event Queue Recovery

Each agent has a message event queue that stores all incoming unprocessed message events for that agent. When a message event is received by an agent and it is doing another job and cannot process the message, the agent pushes the message in a queue and handles it later. When the agent is damaged, this message event queue must be recovered because in fact it represents the assigned responsibilities of the agent.

When the *Supervisor* agent receives the reply for the *request_goal_recovery* message event, it is sure that the goal has been recovered; thus, it starts to recover the message event queue of the damaged agent. The message events of each agent are stored in a snapshot queue corresponding to its unique ID.

This message events snapshot queue is created inside the *ProductionPlan* plan of the *Production* agents. In this plan, after finishing the production task, the *Production* agent calls the existing *Carry* agents by creating and sending the *request_carry* message events to them. At the same time, the created *request_carry* message objects are pushed into the message event snapshot queue of each of the *Carry* agents.

On recovery, the *Supervisor* agent takes this message event snapshot queue and creates a message event for each element stored in the queue and sends it to the new *Carry* agent. More clearly, for the *request_carry* message event, the *Supervisor* agent restores the message snapshot from the queue, creates a corresponding message event and assigns the restored *RequestCarry* object as its content and sends it to the new *Carry* agent. This operation is repeated for all elements of the message event queue snapshot and simulates the copying of the restored message event queue to the new *Carry* agent's message event queue.

By restoring the message event queue, the recovery task is completed and the agent can continue its normal process. Although there may be some message events assigned to the new agent which has not been processed yet, but they become the responsibility of the new agent and will be handled with priority.

## 4.7. Fault-Tolerance of Marsworld in Other Levels

The fault tolerance in group level demands that the system is capable of recovering all of the group members. For each of the agents in the group there is a recovery plan in the *Supervisor* agent. The *RecoverProductionPlan* plan handles the recovery of *Production* agents and the *RecoverSentryPlan* plan recovers the *Sentry* agents.

The *RecoverProductionPlan* plan similar to the recovery plan of the Carry agent, creates a new *Production* agent, recovers its location, recovers its goal in hand that is *produce_ore* goal and recovers its message event queue after receiving the goal recovery confirmation reply from *Production*. In the *Production* agent there are the corresponding *RecoverPlan, RecoverLocationPlan* and *RecoverGoalPlan* plans that fulfill the job.

For the *Sentry* agent the recovery plan is *RecoverSentryPlan* plan on the *Supervisor* side to handle the static recovery, the recovery of location, goal and message event queue. On the *Sentry* side there are the corresponding *RecoverPlan, RecoverLocationPlan* and *RecoverGoalPlan* plans.

Although the recovery of each agent in the group is similar to each other, the goal recovery part can be very different from agent to agent. In fact the most important and complicated recovery plan is that of the goal. The sequence diagrams of the goal algorithm can be used to divide it to smaller tasks and register checkpoints to accomplish the recovery process. Figure 4.13 depicts a snapshot of the recovery scenario in the Marsworld case study. This snapshot shows the moment that the user has clicked on *Carry_3* agent. This user action

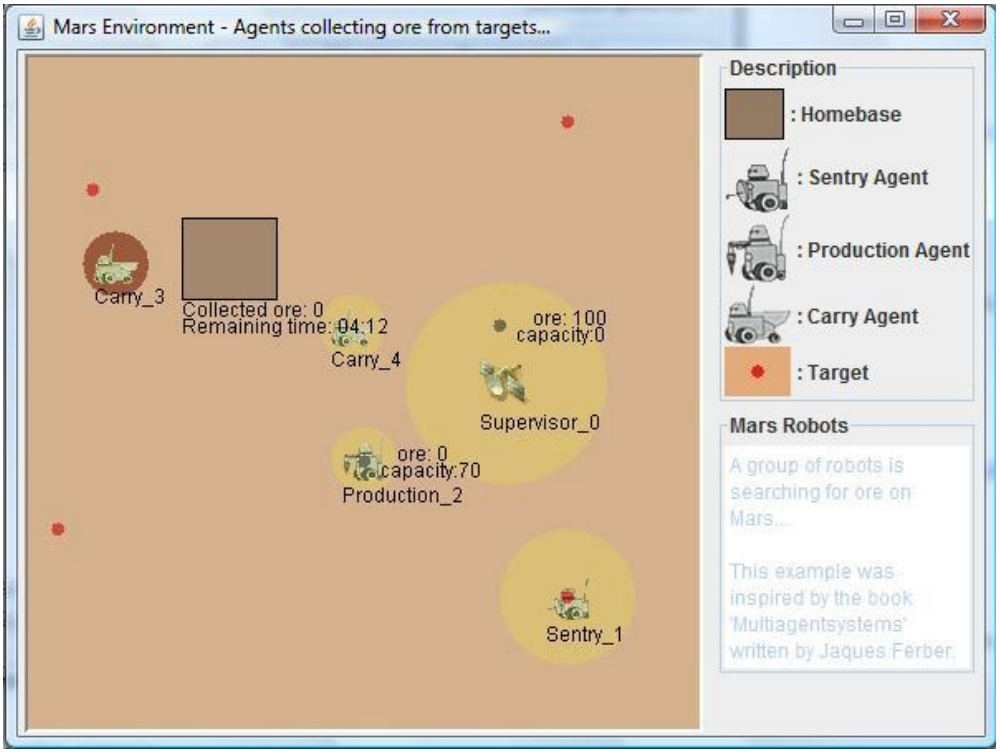


**Figure 4.13:** A snapshot of the recovery scenario of the Marsworld case study.

has caused the deactivation of the *Carry_3* agent. Subsequently, the *Supervisor_0* agent has started the recovery plan that has created the *Carry_4* agent to fulfill the fault-tolerance property. The newly created *Carry_4* agent finishes the recovered tasks of *Carry_3* agent and continues to function as a *Carry* agent in the group to terminate the mission successfully.

## 4.8. Replacement instead of Creation

Until now in this case study, it is considered that agents can be easily created at any time, but in the real world, the system has to take advantage of only the existing *robots* (agents) in the exploration area and their availability has to be taken into account. For example, if a *Carry* agent crashes, the *Supervisor* agent has to find the most available *Carry* agent

instead of simply creating a new *Carry* agent and assign the task to it. This availability can be defined in terms of the location of the agent, either the agent is idle or doing something and the time schedule that it finishes its task.

By considering those parameters, the *Supervisor* agent can choose the proper agent to replace the damaged one and assign the recovery task to it. After choosing the agent, first of all, the *Supervisor* agent waits for that agent until it achieves any incomplete goal in hand. The selected agent can have any position thus the location recovery is different. The *Supervisor* agent may ask the selected *Carry* agent to move to the recovered location by creating *move_destination* goal in *RecoverLocationPlan* plan of that *Carry* agent; otherwise, it can ignore location recovery and directly start the goal recovery. The goal recovery can be more or less similar to what we have seen until now. The only difference is when the *Supervisor* agent decides not to recover the location. In this case the *Carry* agent will start goal recovery from its current location. After goal recovery and when the *Supervisor* agent receives the confirmation response from *Carry* agent, it can recover the message event queue.

This replacement is possible if we prove that the selected agent is the same as the crashed one as the substitutability property that is presented in Chapter 2.

## 4.9. Conclusion

In this chapter we explained the Marsworld case study. Marsworld is a Mars exploration simulation program developed in Jadex that consists of five different types of robots (agents): *Manager*, *Supervisor*, *Sentry*, *Production*, and *Carry*. When the program is

started, these agents are assigned to an exploration area containing ore mines to be exploited.

The objective in this chapter has been the implementation of fault-tolerance property of RAS model with Marsworld case study. To fulfill this objective, recovery plans for each agent type is provided using the RAS architecture and behavior. Also a crash simulation plan is embedded in each agent to test and analyze different scenarios of Marsworld case study.

We have run this program with different scenarios, i.e., with different number and type of agents crashed, and in each case the result of the mission is what was expected. In each test the crashed agent was successfully recovered and the whole system continued its mission correctly to the end. On the other hand, comparing to the four conditions of the substitutability property (see Chapter 2, Section 2.7), we notice that all are satisfied by the recovery plan. The first condition is met because the same type of agent is chosen. For the second condition the two agents have the same message event structure defined in the ADF. Since the replaced agents have exactly the same internal structure and the belief base is restored from log information, the third condition is also fulfilled. Finally to satisfy the last condition, the recovery plan recovers the event queue (INTERACTION) and current goal (TRANSITION) of the crashed agent.

# Chapter 5: Related Work

In this chapter, we review the related work to multi-agent systems, reactive autonomic systems and model transformation.

## 5.1. Multi-Agent Systems for Autonomic Computing

Some of the related work published in the literature that uses the multi-agent technology to implement the autonomic systems can be summarized as follows:

In [25], the authors have developed a distributed software architecture called Unity for autonomic systems based on multi-agent components known as autonomic elements. Unity addresses the achievement of self-management properties such as self-configuration, self-healing, and self-optimization in a dynamic multi-application environment. This paper illustrates the self-configuration of Unity elements at runtime initialization, their method to accomplish recovery from some specific faults as well as management of computational resources among them.

In [27] and [28], the authors have introduced Rudder, a peer to peer agent framework to support autonomic applications in distributed environments. In [27], the focus is on the flexible interaction between agents in systems that are logically decentralized, physically distributed. The peer agents in Rudder use specific protocols to discover, coordinate and

control distributed elements in decentralized environments. These distributed cooperating agents use negotiation strategies to decide and enact the most appropriate adaptation plans. In this paper a peer to peer scalable coordination space called COMIT is proposed to provide communication abstractions.

In [28], Rudder is proposed to support autonomic applications that continuously interact with the environment and with each other to manage their execution in pervasive Grid environments. This management task consists of monitoring, adaptation and optimization of the execution that demands effective coordination services. This paper presents the architecture and operation of Rudder to support the autonomic applications. These applications take advantage of Rudder to coordinate autonomic components and adapt to the requirements and context changes.

The authors in [30] have developed an Autonomic Information System (AIS) by adopting a multi-agent approach. The information system provided by AIS accommodates its processing algorithms and/or information sources to provide necessary information in different efficiency levels. This paper illustrates the accomplishment of certain self-* properties of autonomic systems in AIS and compares it with non-autonomic systems to evaluate its performance. The paper [30] states that AIS is based on the Organization Model for Adaptive Computational Systems (OMACS) [31], which provides the information needed to develop self-organization property of autonomic systems and also allows the reuse and systematically production of autonomic application.

The author in [35] has proposed autonomic computing as a solution for cost-effective and efficient telehealth systems in high demanding health care domain. In order to develop autonomic architectures for telehealth systems, this paper uses a multi-agent

approach. One example of telehealth systems that is presented in this paper to be developed by a multi-agent autonomic system architecture is telemonitoring. Telemonitoring is a system that monitors continuously the health conditions of patients in post-surgery and patients with chronic diseases or life-threatening health problems. This paper states the importance of self-management property for health care systems to justify the importance of autonomic systems in this domain.

In [36], the authors have described two prototype agent-based systems developed at NASA Goddard Space Flight Center (GSFC). These two systems, the Lights-out Ground Operations System (LOGOS) and the Agent Concept Testbed (ACT) address the use of consultations and swarms of nanosatellites that decrease project costs but may cause long delays in communications and loss of contact with the ground control station. The paper [36] discusses the agent-based architecture of LOGOS and ACT, which may be the future of space flight missions. The authors in paper [36] present one scenario example for each of the proposed agent-based architectures and illustrate the self-configuration, self-optimization, self-healing, and self-protection properties of autonomic agents in the examples.

The publication [38] surveys the main approaches for fault-tolerance in multi-agent systems: redundancy (replication) and exception handling. This paper states that redundancy is a method to tolerate faults and errors in components of multi-agent systems. However, to implement the fault-tolerance in multi-agent systems, the designer must take into account not only the cost of initial redundant components, but also the increasing expenses of their maintenance. The paper illustrates exception handling as an

error prone process, which implicates some programming workload. It explains the future direction of exception handling towards automatic error diagnosis beyond tolerance limits.

All the related work presented so far has supported the idea that the multi-agent systems are the appropriate solution for autonomic systems, which justifies the choice of MAS for refining RAS. However, these solutions do not propose a clear mapping from autonomic systems to multi-agent systems. The well-defined architecture of RAS allowed for proposing one to one mapping between RAS components and MAS components. This mapping is used to develop a model transformation approach that automates the transformation of the abstract RAS meta-model to the implementable MAS meta-model (see Chapter 3, Section 3.5).

## 5.2. Agent Programming Tools for Autonomic Systems

In this thesis, Jadex, a Java-based agent programming language is selected as the implementation tool for MAS. However, other development tools and programming languages have been used to implement intelligent agent-based systems:

In [29], the authors have presented the IBM Agent Building and Learning Environment (ABLE), a toolkit for developing multi-agent autonomic systems. This toolkit consists of a lightweight Java agent framework, a comprehensive JavaBeans library of intelligent software components, a set of tools for testing and development, and an agent platform that provides a set of services for ABLE agents. The paper proposes the ABLE distributed agent platform to convince how new features and capabilities can be added to autonomic systems. Using three case studies, the paper explains: 1) the

Autotune agent: a closed-loop controller agent; 2) the Subsumption agent: an agent for specific behaviors and strategies; and 3) the Autonomic agent: an agent with sensors and effectors for interacting with the environment, other Subsumption agents, and other autonomic components in the system. These components form the dynamic model of the autonomic system, its environment, emotions, planning, and executive-level decision-making.

In [32], the authors have proposed an infrastructure called Multi-Agent system based Autonomic Computing Environment (MAACE) for autonomic computing. MAACE is a multi-agent-based architecture based on two previous proposals from the authors: an Infrastructure for Managing and Controlling Agent Cooperation (IMCAC) [33] and an Infrastructure for Managing and Controlling the Social Behavior of Agents (IMCSBA) [34]. The authors in paper [32] state the advantage of using the MAACE environment to manage and control software systems using multi-agent solutions. This environment provides dynamically programmable control and management services by J2EE, CORBA, and .NET technologies to develop intelligent applications. These services include agent federation, agent mediate and agent monitoring. The MAACE infrastructure is used to support the self-configuration and self-healing of network-centric applications.

The authors in [37] have proposed a model of adaptive agent based on well-defined reusable components in order to simplify autonomic systems development. This model implements the non-functional mechanisms such as communication, mobility or adaptation skills by taking advantage of these reusable components. The adaptive agent matches to its runtime environment by changing its components dynamically and autonomously. This improves the safety and performance specifically in open, pervasive,

or large-scale distributed applications. In [37], the authors have presented a tool called Agent$^{\varphi}$ for adaptive agent modeling. This tool consists of a set of operating micro-components, a graphical modeler, an architecture generator (in Java) and a tool for minimization of the architecture.

Regarding the tools and languages used in the discussed related work, Jadex seems an appropriate choice for this thesis. Jadex has many advantages over the other development tools and frameworks such as its flexibility thanks to its XML-based format to define agents and its Java-based format to develop plans for agents (see Table. 2.1 in Chapter 2). The flexible BDI-based architecture of Jadex is another strong reason to give preference to it over other agent programming tools to develop intelligent autonomic systems. Besides, its XML-format Agent Definition File (ADF) has encouraged us to use it as a proper meta-model for our model transformation approach.

## 5.3. Model Transformation

Our research proposes a model transformation approach to transfer the RAS meta-model to the MAS meta-model. There are many model transformation tools that provide different techniques according to their input and output models. Following is a brief discussion about some of these approaches:

In [42], the authors have proposed a classification dividing different models in model transformation area into abstract space and concrete space models. For instance, XML can be classified as the unique concrete level representation technology, whereas UML may be grouped as the abstract level representations. This paper states that model

transformation is an important paradigm in many areas and should be dealt with both at the abstract and concrete levels. The authors of agree that XSLT as a transformation tool in concrete space has to play a central role and more tools must be developed in abstract level that define translation schemes from these languages to XSLT. The paper shows that the problem is more simplified by defining a common meta-meta-model for all models in abstract space which is based on canonical XML transformation. The MOF and XMI standards of Object Management Group (OMG) are used to illustrate this approach.

In [43] the authors have introduced MTRANS project, a general framework for model transformation. The authors in this paper try to keep the MTRANS framework the most general possible, by using the meta-modeling approach, which defines the semantics of each model. MTRANS uses XSLT to transform models, but establishes an abstraction level above XSLT, which is easier to understand. According to this paper, MTRANS supplies a language, which is composed by a fixed instruction set, plus a part depending on the meta-models used. MTRANS can be used to transform MOF compliant models.

Considering the above related work in model transformation and because both our input and output models are in XML format, XSLT seems to be a convenient choice for implementing our model transformation approach. Moreover, the XML standard and XSLT framework have already started to grow, industry wide. As an example, we can refer to Microsoft BizTalk Server that takes advantage of XSLT transformation to provide a solution that allows organizations to more easily connect disparate systems.

# Chapter 6: Conclusion

This thesis aimed at implementing Reactive Autonomic System (RAS) models with Multi-Agent System (MAS) models and introducing a model transformation framework for this purpose. We proposed our approach with the purpose of providing solutions for the following research questions:

1. How can we refine the RAS models in terms of self-* properties into MAS models?
2. What is the appropriate MAS architecture and development agent programming tool to implement RAS?
3. How can we propose a model transformation approach to transform the RAS meta-models to the MAS meta-models?

## 6.1. Contributions

This thesis proposed an automatic refinement of Reactive Autonomic Systems (RAS) models with Multi-Agent Systems (MAS) models and the development of a model transformation framework that establishes/generates the required MAS agent templates from RAS. Besides, this work focuses on Jadex BDI-based agent programming tool to

specify and implement the fault-tolerance property of RAS. The main contributions of this thesis are summarised below:

1. Mapping the RAS static components such as Reactive Autonomic Objects (RAO) to the corresponding MAS components such as agents [Chapter 3].
2. Mapping the behavioral model of RAS to behavioral components in MAS such as *goals* and *plans* [Chapter 3].
3. Defining an Extended BNF grammar to capture the RAS static and dynamic models in XML format [Chapter 3].
4. Proposing a model transformation framework to transform the RAS components to agent templates in MAS [Chapter 3].
5. Specification and implementation of fault-tolerance property of RAS using BDI architecture and Jadex BDI-based agent programming tool [Chapter 4].

## 6.2. Discussions

The architecture of the RAS meta-model has a layered structure consisting of the following components, starting from the lower most primary layer to the top composite layer: *RAO*, *RAC*, *RACG*, and *RAS*. In the lowest layer, there is Reactive Autonomic Object (RAO), which is the atomic component in RAS. Other components in RAS are composite structures built from one or more RAO components communicating with each other. On the other hand, the most primitive component in the MAS meta-model that acts as an autonomic structure is the intelligent agent. In this thesis, we have mapped the RAO component to the agent in MAS. In this mapping, for each RAO an Agent Definition File (ADF) in XML format representing the agent will be generated. This ADF file contains

the definition of building blocks of the agent such as beliefs, goals, events, and plan headers in Jadex BDI-based format. To accomplish the mapping process for the composite elements of RAS, they are decomposed to the atomic RAO with the corresponding communicational link between them. In this case, we create one agent for each RAO having their communicational structure defined as message events in the ADF file for each agent.

The behavioral model of RAS is presented with sequence diagrams. These sequence diagrams show the interactions between different components in RAS. The sequence of interactions in these diagrams is converted to consecutive message triples consisting of *Sender*, *Message*, and *Receiver* in XML format. This stream of messages called *Execution Path* is used to capture the behavior of RAS. In comparison, the behavior of MAS is defined by *plans* that are programs in Java. Each agent can have different plans that are triggered from different sources such as external messages or internally defined goals. The aim is to define plans and their triggering sources (messages, goals) according to the execution paths of RAS. Since the RAS model must have the self-* properties of autonomic systems such as self-healing, the generated MAS component templates from the sequence diagrams for each self-* property will reflect the characteristics of the converted RAS model.

In this thesis, among all agent architectures, the Belief-Desires-Intentions (BDI) model is chosen as the implementation framework. Jadex, as a powerful agent-programming tools based on BDI architecture serves as the development software. The Jadex components are based on the Java programming language, which provides a library that can be used in any Java programming IDE such as Eclipse. The communication

protocol in Jadex is based on FIPA, which is a common agent communication protocol. This thesis uses Jadex to implement the fault-tolerance property in Marsworld [5] case study. Using Jadex library, this case study defines the mapped RAS components as agents and shows that the substitutability property of RAS components can guarantee fault-tolerance.

The RAS meta-model definitions are presented in diagrams and graphical representations that cannot be used in current standard model transformation tools. To capture the static and dynamic aspects of RAS in XML format, this thesis has developed and defined an extended BNF grammar. Using this grammar, we can manually create XML description of each RAS component such as RAO, RAC, etc. The advantage of this grammar is that it allows the automation of RAS transformation to MAS by developing tools that capture the graphical model of RAS as input and create the model in XML as output.

XML is used as a language to describe both the RAS and MAS models. The standard and flexible format of XML permits us to use it in model transformation tools. This thesis proposes a model transformation framework that gets the RAS XML-based model and transforms it to a Jadex-based MAS model. This framework is based on Extensible Stylesheet Language Transformation (XSLT), which is an XML-based model transformation tool. The transformation rules of this model convert the static architecture of RAS (the RAS components to MAS agents) as well as the dynamic behavior (the fault-tolerance sequence diagrams of RAS to the plan templates of MAS). The generated multi-agent templates in Jadex reflect the anticipated fault-tolerant behaviors.

## 6.3. Future Work

This thesis is about the implementation of Reactive Autonomic Systems (RAS) model with Multi-Agent Systems (MAS) model, which opens the door to several research opportunities. Future research may include extending our method to other reactive autonomic systems in industrial scale. In addition, the following orientations could be considered in the future:

- Our implementation focuses on fault-tolerance property of reactive autonomic systems. Whereas, the self-* properties of autonomic systems include self-configuration, self-optimization, self-protection, etc., a future work would be the extension of our model to these properties.
- The process of the RAS model generation from the defined EBNF-based grammar is done manually in this thesis. One direction for future work is the automation of XML-based description of the RAS model. This tool would get the different graphical representations of RAS as input and create automatically the XML files from it.
- Another tool that would be interesting to develop is an IDE to design the RAS model components graphically. This tool can simplify the design process of the RAS model and also can embed the tools to create the RAS XML-based files.
- Our fault-tolerance model is based on substitutability property of the RAS components. Several negotiation strategies for different interests could be added in the future for more sophisticated interactions. Moreover, the semantics of the

recovery plans to have more efficient fault-tolerance techniques should be developed.

- There are some graphical modeling environments for multi-agent systems such as Agent Modeling Language (AML). Some add-ins of notations could be added to this modeling language in terms of reactive autonomic agent. This special kind of agent could act as an intelligent agent with the self-* properties of reactive autonomic systems.

# References


[1] M. G. Hinchey, C. A. Rouff, J. L. Rash, and W. F. Truszkowski, "Requirements of an integrated formal method for intelligent swarms", Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems, Lisbon, Portugal, September 2005, 125-133.

[2] H. Kuang, O. Ormandjieva, S. Klasa, N. Khurshid, and J. Bentahar, "Towards specifying reactive autonomic systems with a categorical approach: a case study", Studies in Computational Intelligence, Volume 253/2009, Springer Berlin/Heidelberg, November 2009, 119-134.

[3] O. Ormandjieva and J. Quiroz, "Methodology for automatic generation of exhaustive behavioral models in reactive autonomic systems", Proceedings of the International Conference on Software Engineering Theory and Practice, Orlando, Florida, USA, July 2008.

[4] H. Kuang and O. Ormandjieva, "Self-monitoring of non-functional requirements in reactive autonomic system framework: a multi-agent systems approach", Proceedings of the 3rd International Multi-Conference on Computing in the Global Information Technology, Athens, Greece, July 2008, 186 – 192.

[5] J. Ferber, "Multi-agent systems: an introduction to distributed artificial intelligence", Addison-Wesley, 1999.

[6] O. Ormandjieva, I. Hussain, "Towards Automatic Generation of Formal Scenarios Specifications from Real-Time Reactive Systems Requirements Written in NL", Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, USA, June 2006, 991 – 999.

[7] V. Wiels and S. Easterbrook, "Management of Evolving Specifications Using Category Theory", Proceedings of the 13th IEEE International Conference on Automated Software Engineering, October 1998, Page 12 – 21.

[8] J. L. Fiadeiro and T. Maibaum, "A Mathematical Toolbox for the Software Architect", Proceedings of the 8th International Workshop on Software Specification and Design, Schloss Velen, Germany, March 1996, 46 – 55.

[9] M. Wooldridge, "An Introduction to Multi Agent Systems", John Wiley & Sons, June 2002.

[10] W. Wan, "Specifying and Verifying Communities of Web Services Using Argumentative Agents", Master Thesis, Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada, August 2008.

[11] C. Vermeulen and B. Bauwens, "Software Agents Using XML for Telecom Service Modeling: a Practical Experience", Proceedings of the SGML/XML Europe'98, May 1998, Page 253 – 262.

[12] P. Maes, "Situated Agents Can Have Goals", *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, MIT Press, February 1991, Page 49 – 70.

[13] A. Pokahr, L. Braubach, "Jadex User Guide", Distributed Systems Group, University of Hamburg, Germany, Release 0.96, 2007.

[14] J. O. Kephart, D. M. Chess, "The Vision of Autonomic Computing", Computer, Volume 36, No. 1, January 2003, Page 41 – 50.

[15] A. S. Rao and M. P. Georgeff, "An Abstract Architecture for Rational Agents", Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning, October 1992, Page 439 – 449.

[16] M. Bratman, "Intention, Plans, and Practical Reason", Harvard University Press, November 1987.

[17] Lemos, R., "ICSE 2003 WADS Panel: Fault Tolerance and Self-Healing", Proceedings of the ICSE 2003.

[18] Chris Inacio, "Software Fault Tolerance", Carnegie Mellon University, 18-849b Dependable Embedded Systems, Spring 1998, http://www.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/ (last checked on 2011-02-22).

[19] H. Kuang, J. Bentahar, O. Ormandjieva, N. Shafieidizaji, and S. Klasa, "Formal Specification of Substitutability Property for Fault-Tolerance in RASF", International Conference on Software Methodologies, Tools and Techniques, 2010, Frontiers in Artificial Intelligence and Applications, IOS Press, pp. 357-380.

[20] Workshop on Domain Specific Visual Languages; "Results Poster". Held at OOPSLA 2001 (organizers: Tolvanen, Gray, Kelly and Lyytinen) http://w3.isis.vanderbilt.edu/OOPSLA2K1/Presentations/Presentations.htm/ ResultsOOPSLA-DSVL-2001.ppt (last checked on 2011-03-15).

[21] S. Sendall, W. Kozaczynski, “Model Transformation - the Heart and Soul of Model-Driven Software Development”, IEEE Software, vol. 20, no. 5, September/October 2003, pp. 42-45.

[22] A. G. Ganek, T. A. Corbi, “The Dawning of the Autonomic Computing Era”, IBM Systems Journal, Volume 42, No. 1, January 2003, Page 5 – 18.

[23] S. Awodey, “Category Theory”, Oxford University Press, USA, July 2006.

[24] ISO/IEC 14977:1996(E), “Information technology — Syntactic metalanguage — Extended BNF”, ISO/IEC, 1996.

[25] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White, “A Multi-Agent Systems Approach to Autonomic Computing”, Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multi-Agent Systems, July 2004, Page 464 – 471.

[26] A. Pokahr, L. Braubach, and W. Lamersdorf, “Jadex: a BDI Reasoning Engine”, Multi-Agent Programming, Springer, September 2005, Page 149 – 174.

[27] Z. Li and M. Parashar, “A Decentralized Agent Framework for Dynamic Composition and Coordination for Autonomic Applications”, Proceedings of the 16th International Workshop on Database and Expert Systems Applications, August 2005, Page 165 – 169.

[28] Z. Li and M. Parashar, “Rudder: A Rule-Based Multi-Agent Infrastructure for Supporting Autonomic Grid Applications”, Proceedings of the 1st International Conference on Autonomic Computing, May 2004, Page 278 – 279.

[29] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao, "ABLE: A Toolkit for Building Multi Agent Autonomic System", IBM Systems Journal, Volume 41, No.3, September 2002, Page 350 – 371.

[30] W. H. Oyenan and S. A. DeLoach, "Design and Evaluation of a Multi Agent Autonomic Information System", Proceedings of the 2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, November 2007, Page 182 – 188.

[31] S.A. DeLoach, W.H. Oyenan, "An Organizational Model and Dynamic Goal Model for Autonomous, Adaptive Systems", Multiagent & Cooperative Robotics Laboratory TR MACR-TR-2006-01. Kansas State Univ. March 2006.

[32] J. Hu, J. Gao, B. Liao, and J. Chen, "Multi-Agent System Based Autonomic Computing Environment", Proceedings of the 3rd International Conference on Machine Learning and Cybernetics, August 2004, Page 105 – 110.

[33] Hu Jun, Ciao Ji, Liao Bei-sbui, Chen Jiu-jun. "An Infrastructure for Managing and Controlling Agent Cooperation", Proceedings of The Eighth International Conference on CSCW in Design, May 26-28.2004, Xiamen, PR China.

[34] Gao Ji, Yuan Chengxian, and Wang Jmg. "IMCSBA: An Infrastructure for Managing and Controlling the Social Behavior of Agents", Chinese J. computers, 2004.

[35] G. Pour, "Prospects for Expanding Telehealth: Multi-Agent Autonomic Architecture", Proceedings of the International Conference on Computational Intelligence for Modeling Control and Automation, and International Conference on

Intelligent Agent, Web Technologies and Internet Commerce, November 2006, Page 130 – 135.

[36] W. Truszkowski, J. Rash, C. Rouff, and M. Hinchey, “Some Autonomic Properties of Two Legacy Multi-Agent Systems – LOGOS and ACT”, Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, May 2004, Page 490 – 498.

[37] S. Leriche and J. P. Arcangeli, “Flexible Architectures and Agents for Adaptive Autonomic Systems”, Proceedings of the 4th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems, March 2007, Page 99 – 106.

[38] Briot, J.P., Aknine, S., Alvarez, I., Guessoum, Z., Malenfant, J., Marin, O., Perrot, J.F., Sens, P.: “Multi-agent systems and fault-tolerance: State of the art elements”, Technical report, LIP6 & MODECO-CReSTIC, (2007) Bibliographic Study.

[39] D. C. Verma, S. Sahu, S. Calo, A. Shaikh, I. Chang, and A. Acharya, “SRIRAM: A scalable resilient autonomic mesh”, IBM Systems Journal, Volume 42, No.1, January 2003, Page 19 – 28.

[40] J. Park, J. Jung, S. Piao, and E. Lee, “Self-healing Mechanism for Reliable Computing”, International Journal of Multimedia and Ubiquitous Engineering, Vol. 3, No. 1, January, 2008.

[41] Liu, H., Parashar, M.: “Accord: a programming framework for autonomic applications”, IEEE Transactions on Systems, Man, and Cybernetics 36(3), 341–352 (2006).

[42] M. Peltier, F. Ziserman, and J. Bézivin, “On Levels of Model Transformation”, XML Europe, Paris, France (2000), pp. 1–17, Graphic Communications Association, 2000.

[43] M. Peltier, J. Bézivin, and G. Guillaume. “MTRANS: A general framework based on XSLT for model transformations”, In WTUML’01, Proceedings of the Workshop on Transformations in UML, Genova, Italy, April 2001.

[44] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. “Multi-Agent Programming: Languages, Platforms, and Applications”. Number 15 in Multi-agent Systems, Artificial Societies, and Simulated Organizations. Springer-Verlag, 2005.

[45] Jeni Tennison. “Beginning XSLT 2.0: From Novice to Professional”. Apress, 2nd edition, 2005.

# Appendix: XSLT transformation example

**Table App 1.** A sample XSLT program source code:

```
<?xml version="1.0"?>
<xsl:stylesheet version="2.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xs="http://www.w3.org/2001/XMLSchema" exclude-
result-prefixes="xs">

    <xsl:output method="xml" indent="yes" version="1.0" />

    <xsl:template match="/">
      <agent xmlns="http://jadex.sourceforge.net/jadex"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >

        <xsl:attribute name="name">
          <xsl:value-of select="RAC/MEMBER/@name" />
        </xsl:attribute>

        <xsl:attribute name="package">
          <xsl:value-of select="concat('marsworld.',/RAC/@name)" />
        </xsl:attribute>

        <xsl:text>
</xsl:text>
        <imports>
          <import>jadex.adapter.fipa.*</import>
          <import>jadex.runtime.*</import>
          <import>jadex.planlib.*</import>
          <import>java.util.Stack</import>
        </imports>

        <events>
          <xsl:text>
</xsl:text>
          <xsl:for-each select="/RAC/INTERACTIONS/INTERACTION">
            <xsl:if test="@source = /RAC/MEMBER/@name">
              <messageevent name="{@name}">
                <xsl:attribute name="type">
                  <xsl:value-of select="'fipa'" />
                </xsl:attribute>
                <xsl:attribute name="direction">
                  <xsl:value-of select="'send'" />
```

```
                </xsl:attribute>
                <xsl:text>
  </xsl:text>

                <parameter name="performative" class="String"
direction="fixed">
                  <xsl:text>
    </xsl:text>
                  <value>SFipa.REQUEST</value>
                  <xsl:text>
  </xsl:text>
                </parameter>
              </messageevent>
            </xsl:if>

            <xsl:if test="@target = /RAC/MEMBER/@name">
              <messageevent name="{@name}">
                <xsl:attribute name="type">
                  <xsl:value-of select="'fipa'" />
                </xsl:attribute>
                <xsl:attribute name="direction">
                  <xsl:value-of select="'receive'" />
                </xsl:attribute>
                <xsl:text>
  </xsl:text>
                <parameter name="performative" class="String"
direction="fixed">
                  <xsl:text>
    </xsl:text>
                  <value>SFipa.REQUEST</value>
                  <xsl:text>
  </xsl:text>
                </parameter>
              </messageevent>
            </xsl:if>
            <xsl:text>
</xsl:text>
          </xsl:for-each>

        </events>
      </agent>
    </xsl:template>
</xsl:stylesheet>
```

**Table App 2.** The input XML file example:

```
<?xml version="1.0" encoding="UTF-8"?>
<RAC name = "rac-name">
  <MEMBER name = "CU1"/>
  <INTERACTIONS>
```

```
    <INTERACTION source="CU1" name="restart" target="Sensor1"/>
    <INTERACTION source="Sensor1" name="heartbeat" target="CU1"/>
    <INTERACTION source="CU1" name="request_sensor" target="CU8"/>
    <INTERACTION source="CU1" name="register" target="Sensor8"/>
    <INTERACTION source="CU1" name="take_over_sensor" target="Drill1"/>
    <INTERACTION source="Drill1" name="confirm" target="CU1"/>
  </INTERACTIONS>
  <LEADER name = "CU1"/>
</RAC>
```

**Table App 3.** The output ADF file example:

```
<?xml version="1.0" encoding="UTF-8"?>
<agent xmlns="http://jadex.sourceforge.net/jadex"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" name="CU1"
package="marsworld.rac-name">

<imports>
<import>jadex.adapter.fipa.*</import>
<import>jadex.runtime.*</import>
<import>jadex.planlib.*</import>
<import>java.util.Stack</import>
</imports>

<events>
<messageevent name="restart" type="fipa" direction="send">
  <parameter direction="fixed" class="String" name="performative">
    <value>SFipa.REQUEST</value>
  </parameter>
</messageevent>

<messageevent name="heartbeat" type="fipa" direction="receive">
  <parameter direction="fixed" class="String" name="performative">
    <value>SFipa.REQUEST</value>
  </parameter>
</messageevent>

<messageevent name="request_sensor" type="fipa" direction="send">
  <parameter direction="fixed" class="String" name="performative">
    <value>SFipa.REQUEST</value>
  </parameter>
</messageevent>

<messageevent name="register" type="fipa" direction="send">
  <parameter direction="fixed" class="String" name="performative">
    <value>SFipa.REQUEST</value>
  </parameter>
</messageevent>

<messageevent name="take_over_sensor" type="fipa" direction="send">
  <parameter direction="fixed" class="String" name="performative">
    <value>SFipa.REQUEST</value>
  </parameter>
</messageevent>
```

```
<messageevent name="confirm" type="fipa" direction="receive">
  <parameter direction="fixed" class="String" name="performative">
    <value>SFipa.REQUEST</value>
  </parameter>
</messageevent>
</events>
</agent>
```

---