Universität Stuttgart
Institut für Automatisierungs- und Softwaretechnik
Prof. Dr.-Ing. Dr. h. c. P. Göhner

Master Thesis

Pedro Fernando Gomez Fernandez

1911 MT

Development of an Agent Oriented Control Application to a Railway Model

Stuttgart, 9th September 2003

Tutors: **Prof. Dr.-Ing. Dr. h. c. Peter Göhner**
Paulo Urbano, M. Sc.

Abstract

The goal of this project is to create an application to control a railway model using an agent oriented methodology. This will allow us to evaluate the benefits and advantages of agent oriented programming.

This project will be composed of the research of the agent concept and its definitions. The investigation of the different agent oriented methodologies and the application of one of them to the analysis and design of a railway control application will be also considered. At the end the development of a prototype of the program will be realized following the results of the analysis and design phases using the agent oriented methodology.

The prototype will be able to control the trains of the railway model and they will follow a provided timetable; it will have high fault-tolerance and will be able to react to situations not imagined in design time. This prototype will be also useful to analyze the benefits of the agent oriented paradigm.

Table of contents

1 Introduction

1.1 Motivation

The actual methods to develop distributed systems are specially suited to environments where the architecture and the general working conditions and requirements do not change. In such context, the control software can be optimised in many aspects, due to the a priori knowledge about the system and its conditions. On the other hand, in case of unplanned modifications in the expected runtime conditions or failures in one of the systems' elements, only a limited reaction is provided by the system itself due to its static nature. In many cases the system as a whole has to be stopped in order that repairs, reconfigurations and new planning of its execution steps can be made.

The model railway in the Institute of Industrial Automation and Software Engineering (IAS) of the University of Stuttgart is a good example of a distributed system. The model is composed of five different rail stations – for persons, for goods and train parking places – and eight trains. It is possible to control all this elements using PC based software.

The complexity of the control system can be easily demonstrated by the following requirements:

- The use of a specific track has to be exclusively given to a train, otherwise crashes could happen. The global assignment of train lines to trains have to agree with each train's individual timetable.
- Each train has a timetable and tries to keep up with it. Problems arise in case of delayed or broken trains, closed roads or rail stations. Modifications in the trains' timetables have to be made dynamically in order that the overall rail system functionality is not severely affected.
- A centralised solution to this problem can result in a very complex system with difficult maintenance.

There is a research line at IAS that deals with the use of agent oriented software engineering (AOSE) to develop dynamic, complex, distributed software. The agent concept models the system's elements as autonomous units capable of co-operating with each other through a negotiation process, in order to achieve a common goal. The negotiation process can be used to solve failure situations as mentioned above without the necessity of human intervention, reducing the system's downtime and maintenance costs and increasing fault tolerance and reliability.

1.2 Objectives

The goals of this master thesis are: the analysis and design of a control application to the railway model, using an agent oriented development method; the prototypical implementation of agent based software to control the model; and the evaluation of the new features of the developed software in scenarios including breakdown of trains and rail stations and changes in the timetables due to delays.

2 System Requirements

2.1 Mandatory Criteria

The project implies the development of an agent oriented solution for a control application to the railway model at the IAS. It must use Gaia for the design and analysis of the agent oriented solution and use Agent UML as the modelling language for the solution. Each train will have a timetable that can be dynamically modified.

It also requires the realization of a prototype that will control the model railway. The prototype will create agents implemented according to the design; they will be able to interact with other agents. It will be also necessary to send commands to the railway model and get its answers in order to control the trains and turnouts.

At last the project involves the evaluation of new features of the prototype in dynamic scenarios as breakdown of trains and tracks or changes in the timetables because of delays.

2.2 Optional Criteria

A lossy communication channel among agents will be implemented to simulate messages losses and their impact in the agents' performance.

Agents will avoid falling in a blocked state in which they do nothing but waiting for an event to occur. There also will be three different ways in which an agent can react to the arrival of a message; it can process the message immediately when the message arrives, it can wait until the agent is in an appropriate state for receiving the message or it cannot process the message at all.

Agents could be created and destroyed dynamically when the program is running.

2.3 Areas of Application

The most evident application of results of this project is the control of the model railway at the IAS.

But this project will also provide a general package for the development of agents that will be as general and flexible as possible. The most significant restriction of this package will be that agents must run in the same computer and program.

It will be also used as a reference for the design of other agent oriented projects using the Gaia methodology.

It will be useful in the creation of tools that work for the railway model, as it will be flexible and will allow the control of trains in a safe way.

2.4 User Group

Users of this project are persons who work with the model railway, and in the Industrial Automation Laboratory Course.

People who will develop agent oriented projects can also use the results of this work to learn from the analysis and design processes done here.

Also people interested in implementing agents can use the agent package of the project.

2.5 Requirements to the Conception

/R10/ A concept for the creation of an agent based control system should be developed**/LA10/**

/R20/ The analysis of the project must be done using the Gaia methodology. From the Gaia analysis we will get a "Roles Model" and a "Interaction Model" **/LA20/**

> **/R21/** Identify the roles in the system **/LA20/**
>
> **/R22/** For each role, identify and document the associated protocols so an "Interaction Model" is realized **/LA20/**
>
> **/R23/** Elaborate the "Roles Model" using the "Interaction Model" **/LA20/**

/R30/ The design of the project must be done using the Gaia methodology. From the Gaia design we will get an "Agent Model", a "Services model", and an "Acquaintance model" **/LA20/**

> **/R31/** Create an agent model identifying the agent types **/LA20/**
>
> **/R32/** Develop a services model identifying the main services required **/LA20/**
>
> **/R33/** Develop an acquaintance model documenting lines of communication between agents. **/LA20/**

/R40/ The Agent UML must be used as modelling language.**/LA20/**

/R50/ There must be a timetable for each train, and each train must follow it in the closest possible way. If problems arise, the timetable must be changed dynamically by the train./**LA30/**

/R60/ There will be possibly only one train running on each track to avoid collisions./**LA30/**

/R70/ A train must be able to go from its origin station to the destination station in a safe way finding the optimal path for a global profit.

/R80/ A prototype must be implemented to control the model railway. **/LA40/**

/R81/ An agent package must be implemented to allow the creation of general agents that can communicate with each other in a flexible way.

/R82/ Each agent will be implemented according to the design step.

/R90/ The communication with the railway model will be implemented in java.

2.6 Quality Requirements

Product Quality	very high	High	Normal	not relevant
Theory		**X**		
Grade of Abstraction		X		
Consistency		X		
Functionality		**X**		
Correctness		X		
Safety	x			
Security				x
Usability			**X**	
Comprehensibility			x	
Ability to learn			x	
Applicability		**X**		
Analysis possibility		X		
Portability				**x**
Modifiableness			x	

2.7 Execution

The thesis has to be executed according to the “IAS Process Model” (Model for Conceptions).

The state of the thesis and the results have to be discussed with the tutors in a period of 2 weeks.

The IAS guidelines have to be respected.

2.8 Prototype Environment

2.8.1 Software

The prototype will be a Java program which will interact with the model railway by means of a program that communicates with the model through the serial port and with our program through a socket channel.

2.8.2 Hardware

A real model railway will be controlled by the prototype program. The model is controlled by a Intellibox [34] controller. Our program will send commands to the Intellibox in order to control the railway model.

2.8.3 System Interfaces

Our Java prototype will communicate with an interface program through a socket channel. This interface program will forward what receives from our program to the Intellibox through a serial port from the computer.

Messages from the Intellibox will be sent to the interface program through the serial cable to the serial port of the computer. Then the interface program will send the responses to our prototype by means of the socket channel.

The interface program must be in a computer connected to the model, but the prototype can be in any other computer that has internet connection with the first one. The system interfaces diagram can be seen in Figure 2-1.

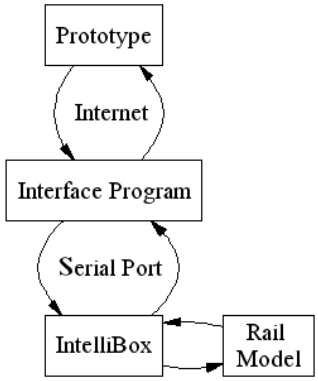

Figure 2-1: System Interfaces

2.9 Quality Requirements for the Prototype

Product Quality	very high	high	normal	not relevant
Functionality		x		
Correctness		x		
Safety	x			
Security				x
Reliability		x		
Maturity			X	
Fault Tolerance		x		
Recoverableness		x		
Usability			X	
Comprehensibility			X	
Ability to learn			X	
Operability			X	
Efficiency			X	
Time Performance			X	
Consumption behavior			X	
Alteration capability		x		
Analysis possibility		x		
Modifiableness		x		
Portability				x

2.10 Development Environment

2.10.1 Software

This prototype will be developed in Java and will use an interface program to interact to the railway model.

2.10.2 Hardware

A PC computer connected to the model railway by means of a serial cable will be used. The prototype program can control the model from any computer connected to the Internet if there is a computer connected to the model and to Internet and which has the interface program running.

2.11 Global Evaluation Methods

Evaluation of the new features of the developed software in scenarios including breakdown of trains and rail station and changes in the timetables due to delays will be done. The evaluation of losses of messages will be also possible in the mentioned scenarios.

3 Basics

3.1 Introduction

The Object Oriented (OO) paradigm has helped indeed in the software development by means of several useful concepts; agent oriented paradigm uses, among others, the concepts that are part of the OO to help in the development of software projects that work with a very dynamic environment with a high degree of uncertainty.

There is a great expectation about the agent oriented paradigm, but it is very recent and there is no standard way to approach to a problem using it. In addition, whereas languages as C++, Java, Python, Lisp, etc implement the OO paradigm, no programming language except experimental ones supports the agent oriented paradigm. Several methodologies have been developed to solve this problem and create a standard way to develop an agent oriented project using OO languages.

I will also explain in this document the agent oriented paradigm as well as the most important methodologies developed to use it. But first I will explain the main OO concepts as they will be used by the agent oriented paradigm.

Therefore, in this document first an overview to the OO paradigm is provided, then an introduction to agents is given, after that an overview of agent oriented methodologies with special emphasis on Gaia will be exposed and then the characteristics that make Java a very suitable language for the develop of agent oriented programs will be described. And finally, there will be an overview of all the exposed in the document.

3.2 Introduction to the Object Oriented Paradigm

Paradigm is defined as: "a set of theories, standards, and methods that together represent a way of organizing knowledge", [2].

Software development has evolved through very different paradigms; each of them represents a new abstraction level.

In the first days of programming, a big monolithic program was the result of any software development; but this became too complex with big programs because their interrelations and dependences.

Nowadays, several abstraction mechanisms exist. They can be grouped as [1]:

Type Abstraction:

- **Basic Data Types:** These are the data types used at first as integers, floating point number and characters.
- **Functions:** Function abstraction helps in grouping code that makes an specific task in functions. So the programmers can concentrate in doing small tasks instead of big ones and allowing them to organize repetitive tasks in one place.
- **Modules:** The module abstraction allows to group different functions that were related and creates and manages namespaces allowing information hiding.
- **Abstract Data Types:** The abstract data types allows the programmers to define data types that can be manipulated as predefined data types
- **Objects:** Are a integration of the abstract data type with the module so, an object package encapsulates the data with the functions that operate over the data.
- **Class**: Are mechanisms by which knowledge about objects can be captured.
- **Generalization/Specialization:** Generalization and specialization are relation between classes that allow them to share the same code.
- **Polymorphism:** Extends generalization and specification to allow the shared code to be customized to fit the specific circumstances of each of the individual classes.
- **Interface:** They are a collection of services without implementation that can be implemented by objects. It is a second type of hierarchical classification.
- **Reflection:** Allows an application to get detailed information about an object.

Service Activation Abstractions:

- **Function Call:** The programmer identifies the function by name, passes the required arguments, and bound the result to an appropriate variable.
- **Event Processing (Asynchronous Communication):** Separates the circumstance in which the need for a service arose from the invocation of the service.

- **Message Passing (Synchronous Communication):** An action is initiated by a service request (message) sent to a specific object.
- **Subscription (Asynchronous Communication):** An object registers with an event handler that it is interested in receiving events handled by it.

Processing Control Abstractions:

- **Single Program Execution:** One program at a time was executed
- **Multitasking**: More than one program can be executed in the same machine at virtually the same time, quasi-parallelism.
- **Sequential Execution:** At every given time during the execution of the program, there is a single thread of execution.
- **Multithreading**: During the execution of the program, several threads are running at virtually the same time

Relationships Abstractions:

- **Associations**: Describes a group of links with common structure and semantics.
- **Aggregation**: This associate an object that represents a whole with a set of objects representing its components.

Behaviors:

- **Static Behavior:** The operation within a method will not be affected by any external or internal events.
- **Dynamic Behavior:** The operations within a method will depend on its state.

Rules: They are mechanisms for specifying the data semantics of any application domain. Most declarative semantics are explicitly specified in rules.

The OO paradigm helps in the structuring and development of very complex systems allowing the use of the earlier mentioned powerful concepts like interfaces, classes, objects, generalization/specification, polymorphism, reflection, message passing, subscription, associations, aggregations, behaviours and rules (this in a very weak way).

The agent oriented paradigm helps in the development of programs that deals with dynamic and stochastic scenarios. This paradigm shares concepts of the OO abstractions and makes agent oriented paradigm more useful for these scenarios, making the agent an acting entity.

3.3 Agents

In order to understand better the agents it is convenient to explain where the agent concept comes from.

In the Artificial Intelligence (AI) development, four approaches can be observed depending of the kind of system we want construct [5]:

- **Systems that think like humans: The cognitive modeling approach.**

  This needs to get inside the actual workings of humans minds. Real cognitive science is necessarily based on experimental investigation of actual humans or animals.

- **Systems that think rationally: The "laws of thought" approach.**

  It is based on the irrefutable reasoning process, on the logic.

- **Systems that act like humans: The Turing Test approach.**

  They must have the following capabilities: natural language processing, knowledge representation, automated reasoning and machine learning.

- **Systems that act rationally: The rational agent approach.**

  It is the more interesting approach for our project because the need of a system that acts in a rational way. It is more amenable to scientific development than the ones based on human behaviour or thought because rationality is clearly described. Even it is more general than "system that thinks". In addition it is the most successful approach in the AI.

The rational agent approach is a very theoretical concept, but the developing of this technique has evolved to the creation of more usable agent definitions and the development of methodologies to deal with these agent definitions.

3.3.1 The Disagreement about the Agent Definition

There is no general agreement about what an agent is.

Here we can see some definitions of the term “agent”:

a) "An agent is a computer system that is situated in some environment and that is capable of autonomous action in this environment in order to meet its design objectives" [3]

   Software agents are the ones that have the following characteristics:

   - They are situated in some environment
   - They are capable of flexible autonomous action in order to meet design objectives. [4]

b) "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors." [5]

c) "Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act" [33]

d) "Intelligent agents continuously perform three functions: perception of dynamic conditions in the environment; action to affect conditions in the environment; and reasoning to interpret perceptions, solve problems, draw inferences, and determine actions." [6]

e) "...a hardware or (more usually) software-based computer system that enjoys the following properties:

   - Autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;
   - Social ability: agents interact with other agents (and possibly humans) via some kind of agent-communication language;
   - Reactivity: agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;
   - Pro-activeness: agents do not simply act in response to their environment; they are able to exhibit goal-directed behavior by taking the initiative." [7]

A list of attributes that an agent may posses is:

- Reactivity/Responsiveness: reacts and responds to changes in environment.
- Autonomy: has the control of its own actions.
- Communicative/Collaborative behavior/Socialability: communications with others agents.
- Proactivity/Goal-orientation: Acts according its objectives.
- Reasoning/Rationality
- Learning: change its behavior based on its previous experience.
- Mobility: able to transport itself from one machine to another.
- Flexibility: actions are not immutably defined.
- Personality/Character/Anthropomorphism
- Temporal continuity: continuously running process independent of external activities.
- Benevolence
- Veracity

And possibly there are more attributes an agent can have.

3.3.2 Autonomous Agent Definition

The great diversity of agents' definitions exists because there is not an accordance of which of the earlier mentioned characteristics an agent must have.

Here it will be defined an agent that will be useful for the development of the kind of systems we are interested, which are automation devices, and to be a very applicable agent.

It is found that agents will need the following characteristics:

- **Reactivity/Responsiveness:** This is needed because the agents need to interact with the environment. This is a characteristic of all agents' definitions.
- **Communicative/Collaborative behavior/Socialability**: This is needed because in most cases in automation devices more than one agent will be needed in the same environment and they will have to communicate with each other to perform a common goal.

- **Autonomy**: This is needed because agents should be autonomous entities in their actions. This is a characteristic of all agents' definitions.
- **Proactivity/Goal-orientation**: Agents will autonomously act in order to reach a goal; they have goals and will act in order to achieve them by their own, not in response to a request. This characteristic is the one that will be needed specifically for automation devices because its need of accomplish a task in a autonomously way.

Reactivity/Responsiveness let the agent to interact with its environment as seen in Figure 3-1.

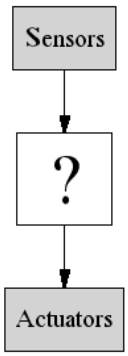

Figure 3-1: Agent Reactivity/Responsiveness

The percepts are the agent's perceptual inputs at any given instant. An agent's percept sequence is the complete history of everything the agent has ever perceived.

Communicative /Collaborative behavior/Sociability let the agent to interact with others agents. The communication method will be based on asynchronous messages that agents can send and receive as seen in Figure 3-2**.**

Figure 3-2: Message sending

Autonomy lets the agent be an independent entity. The agent's behavior is described by the agent function that maps any given percept sequence to an action. The agent function for an agent will be implemented by an agent program.

Proactivity/Goal-orientation points where the agent must conduct its actions. This is a key part of the autonomous agent because the agent knows the goals it must reach and will act by itself in order to accomplish them.

Autonomous Agent Definition:

In the agent oriented software engineering, the concept of an agent, as defined by the IAS (Institute of Industrial Automation and Software Engineering), represents a bounded software entity with a defined goal. An agent tries to reach its goal through an autonomous behavior and interacts continuously with its environment and with other agents.

But "…it must be taken into consideration that an agent is meant to be a tool for analyzing systems, not an absolute characterization that divides the world into agents and non-agents." [5] This means that the agents will be an abstraction used to help the programmer in the analysis and design phases of the development.

3.3.3 Agent Oriented Paradigm Contribution to Object Oriented Paradigm

Like object oriented paradigm unified data and methods adding some valuables concepts, agent oriented paradigm add goals and autonomy to objects making the agents acting objects; and also adds another important concepts.

So the agent will use several of the concepts used by objects which makes the agent more flexible when it is developed.

What is an Object?

From an application modeling perspective, an object has the following components:

- Characteristics or attributes: They are internal information that describes the object
- Services or Behaviors
- Unique Identifier
- Rules and Policies
- Relationships

The agents adds to the object a set of goals and the autonomy required to be an acting entity as well as the ways to interact with its environment and with others agents.

The agent will be probably implemented as an object as no stable languages implements agent paradigm nowadays, but the difference between them is the different abstractions they represent in the analysis and design phases.

3.3.4 The Task Environment

The task environments [5] are the “problems” to which agents are the “solution”. The flavour of the task environment directly affects the appropriate design for the agent program.

The task environment is composed of:

- Performance measure: It allows knowing how well the agent accomplishes its goals.
- Environment: The environment in which the agent is running.
- Actuators: The possible ways an agent can act in the system.
- Sensors: The perceptions the agent collects from the environment.

A small number of dimensions can be identified along which task environments can be categorized, they are:

- **Fully observable vs. partially observable.**

  If an agent's sensor gives it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable.

- **Deterministic vs. stochastic**

  If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic.

- **Episodic vs. sequential**

  If the agent's experience is divided into atomic episodes, then we say the environment is episodic.

- **Static vs. dynamic**

  If the environment can change while an agent is deliberating, then we say the environment is dynamic.

- **Discrete vs. continuous**

  The discrete/continuous distinction can be applied to the state of the environment, to the way time is handled, and to the percepts and actions of the agent.

- **Single agent vs. multiagent**

  In multiagent environments, agent-design problems arise as communication and stochastic behaviour. We can handle competitive or cooperative multiagents environments.

We can identify the task environment of our project as partially observable, stochastic, sequential, dynamic, continuous and cooperative multiagent.

It is partially observable because we have very limited sensors and we do not have the full information of our environment, we must guess what it is happening.

It is stochastic because we must deal with a real railway model that may not work as expected, and modifications to the environment can be introduced by the user at any time.

It is sequential because a decision taken at a certain time is determined by all previous actions.

It is dynamic because the real model is in action meanwhile the agents are taking decisions.

It is continuous because we handle time as continuous and the number of states of the environment is so large that cannot be modelled using current methodologies. The perception of the environment is discrete, but the environment has so many states and we will treat it as continuous.

It is cooperative multiagent because we will use multiple agents that will try to get a collective best outcome.

It is the hardest case, but it is also the most common in real situations.

3.3.5 Autonomous Agents

They are the most suitable agents to our problem domain [5] and can be of interest even in single-agent environments. The most important are:

- **Goal-based agents:**

  The most effective way to handle partial observability is for the agent to keep track of the part of the world it cannot see now. That is, the agent should maintain some sort of internal state that depends on the percept history and thereby reflects at least some of the

unobserved aspects of the current state. We also need some information about how the world evolves independently of the agent and how the agent's own actions affect the world. This knowledge is called model of the world. These agents have a goal information that describes the situations that are desirable. The agent program can combine this with information about the results of possible actions in order to choose actions that achieve the goal. Goal-based agents are flexible because the knowledge that supports its decisions is represented explicitly and can be modified. See Figure 3-3.

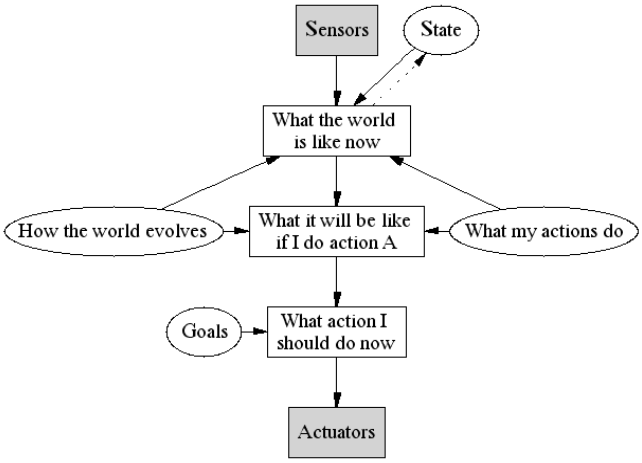

Figure 3-3: Goal-based agent

- **Utility-based agents:**

  A utility function maps a state onto a real number, which describes the associated degree of success. The agent will try to maximize this degree of success. See Figure 3-4.

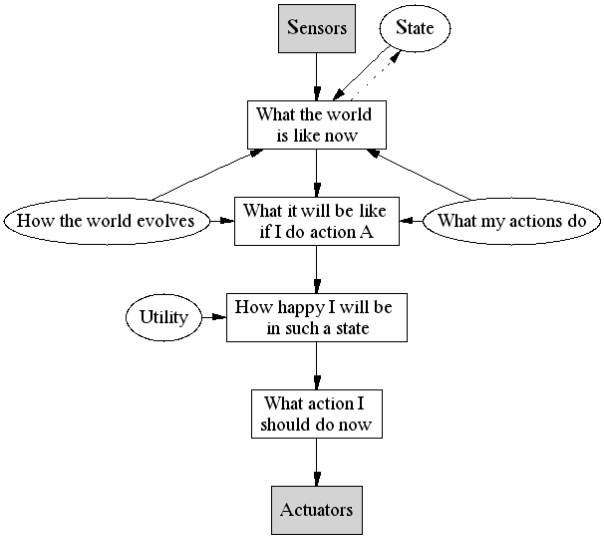

Figure 3-4: Utility-based agent

- **Learning agents:**

  All the previous agents can be converted to learning agents by means of different methods that will try to improve its performance evaluating the accomplishment of the goals using different methods. They will be also able to develop new strategies to reach goals.

3.3.6 Multi-Agent Systems

A multi-agent system in the context of this thesis is one in which:

- Two or more agents exist.
- There are agent communications.

There are several coordination scenarios:

- Cooperation: the agents try to perform a common goal.
  - Planning: Deciding a way to accomplish the task.
    - Distributed Planning: The planning is performed for all the agents.
    - Centralized Planning: The plan is performed by one agent collecting information from other agents.
- Competition: the agents try to perform its own goal.
  - Negotiation: The agents may negotiate a common strategy if it is good for both agents.

The agents can send two different types of messages:

- Assertion: when an agent stays something.
- Query: when an agent ask for resources.

With these different messages all agents can communicate with each others, get data from them and send petitions.

3.4 Agent-oriented Software Development Methodologies

Agent oriented methodologies [8] can be classified in two groups: the ones that evolves from an object oriented methodology like Gaia, AAII or MESSAGE, and the ones that does not derive from an object oriented methodology like PASSI and AOR. In figure 3-1 the evolution of different methodologies can be observed. PASSI and AOR do not derive from any OO methodology. There are four OO methodologies in the diagram: RUP, OMT, Fusion and OPEN and the derived agent oriented methodologies are pointed by arrows from them. The rest of agent oriented methodologies like Prometheus, Cassiopeia, Tropos and MASSIVE does not derive from the rest of methodologies, but have some similar characteristics.

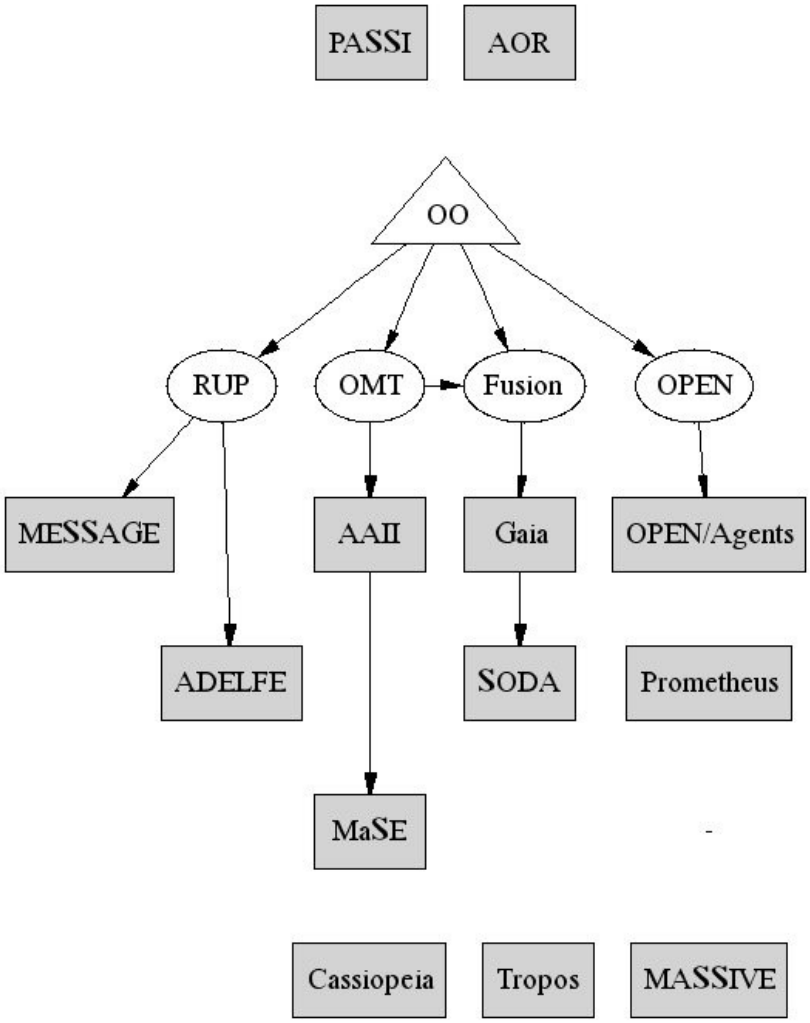

Figure 3-5: Agent methodologies

In this chapter previous methodologies will be explained with special emphasis on the Fusion, Gaia and SODA and OMT, AAII and MaSE branches.

More information about other methodologies as ADEPT, AO, AOR, CoMoMAS, DESIRE, MAS-CommonKADS and Styx and details about Gaia, MaSE, MESSAGE, Prometheus, and Tropos can be found in [9].

3.4.1 The non OO Derived Methodologies

There are some agent oriented methodologies that does not derive from any OO methodology like AOR and PASSI.

3.4.1.1 AOR (Agent-Object-Relationship):

This methodology does not have any processes to derive the agents, events or actions from the analysis of the problem. So it is only well suited when the analysis and design work have been done by others means and the results must be represented.

3.4.1.2 PASSI:

This methodology will build some models that will lead towards the development of a project from its requirements. The analysis and design processes are integrated in the development of these models.

- **System Requirements Model**: it is an anthropomorphic model of the system requirements in terms of agency and purpose.
- **Agent Society Model**: it is a model of the social interactions and dependencies among the agents involved in the solution.
- **Agent Implementation Model**: it is a model of the solution architecture in terms of classes and methods.
- **Code Model**: it is a model of the solution at the code level.
- **Deployment Model**: it is a model of the distribution of the parts of the system across hardware processing units, and this migration between processing units.

3.4.2 OO Derived Methodologies:

An OO methodology uses concepts of classes, class features and a specific set of relationships based on the object client-server model.

The alternative approach in agent oriented methodologies is to take an existing OO methodology and extend it to support agent concepts.

Some OO methodologists identify three roughly chronological "generations" of object modeling techniques [13]:

- In the first generation, isolated methodologists and small groups developed techniques that solved problems they saw first-hand in OO development projects. In this generation are included people and techniques such as Rumbaugh, Jacobson, Booch, CRC, Formal methods, Shaler-Mellor, and Yourdon-Coad.
- The second generation recognized that many best practices were scattered among the fragmented OO methodology landscape. Several attempts were made to gather these practices into coherent frameworks such as **Fusion**. However, the OO community was beginning to recognize the benefits that industry standardization would bring: not just a good way of doing things, but the good way, which would lead to common parlance and practice among developers.
- The third generation consists of credible attempts at the single industry-standard UML (Unified Modeling Language) like **OPEN**.

3.4.3 The Fusion-Gaia-SODA Branch

The Fusion OO methodology and the Gaia and SODA agent oriented methodologies will be explained in the next sections. It is possible to notice how Gaia takes the method of Fusion and adapts it in order to deal with agents. Then SODA methodology modifies Gaia to take the agent space into account but loosing the simplicity of Gaia for closed domain agent systems.

3.4.3.1 Fusion:

The Fusion Methodology [14] is a second-generation OO methodology with the followings benefits:

- The analysis process generates searching questions about requirements so you can find omissions and ambiguities on them.
- Analysis leads to a better understanding of the problem domain, so the models better fulfill the requirements.

- The models and notations used in the analysis and design allow problems to be explored at different levels of abstraction.
- It also will provide a lot of benefits for team development and maintenance by people other than the developers.

The Fusion methodology is composed of the following elements show in Figure 3-2.

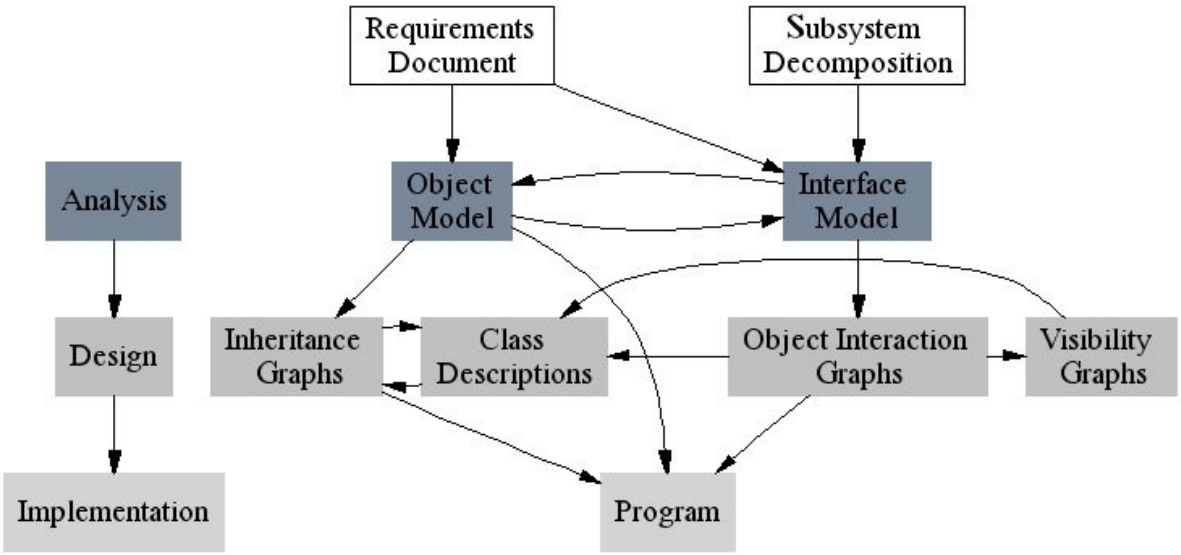

Figure 3-6: Fusion diagram

- **Object Model:**

  It captures the concepts that exist in the requirements document and the relationships between them. It represents classes, attributes and relationships between classes.

- **System Interface:**

  The system interface is the set of system operations to wich a system can respond and the set of events that it can output.

- **Interface Model:**

  It is composed of:

  - The Life-Cycle Model:

Defines the allowable sequences of interactions in which a system may participate.

- The Operation Model

  It defines the meaning of each system operation in the system interface.

- **Interaction Graphs**

  Shows how functionality is distributed across the objects in a system.

- **Visibility Graphs**

  Shows how the object-oriented system is structured to enable object communication.

- **Class Descriptions**

  Specify the internal state and external interface required by each class.

- **Inheritance Graphs**

  It identifies commonalities and abstractions in the classes.

- **Programming**

  It maps the design into effective implementation.

It will be observed lots of similarities between the Fusion and Gaia Models. For example, in the analysis phase, the “Object Model” in Fusion will be similar to the “Roles Model” in Gaia and the “Interface Model” will be similar to the “Interactions Model”. In the Design phase, “Interaction Graphs” will be similar to the “Services Model”, “Visibility Graphs” to the “Aquaintance Model”, and “Class Descriptions” and “Inheritance Graphs” to the “Agent Model”

3.4.3.2 Gaia:

For the analysis and design of our project the Gaia methodology [15] [16] will be used.

Gaia is a general methodology that supports both the micro-level (agent structure) and macro-level (agent society and organization structure) of agent development, but it requires that inter-agent relationships and agent abilities are static at run-time. Using Gaia is possible to systematically develop a design based on system requirements.

3.4.3.2.1 Description

Gaia is a methodology for agent oriented analysis and design. It is founded on the view of a multi-agent system as a computational organization consisting of various interacting roles.

In Gaia there are two main steps: analysis and design. In each of them some models will be created. A diagram of this process is shown in Figure 3-3.

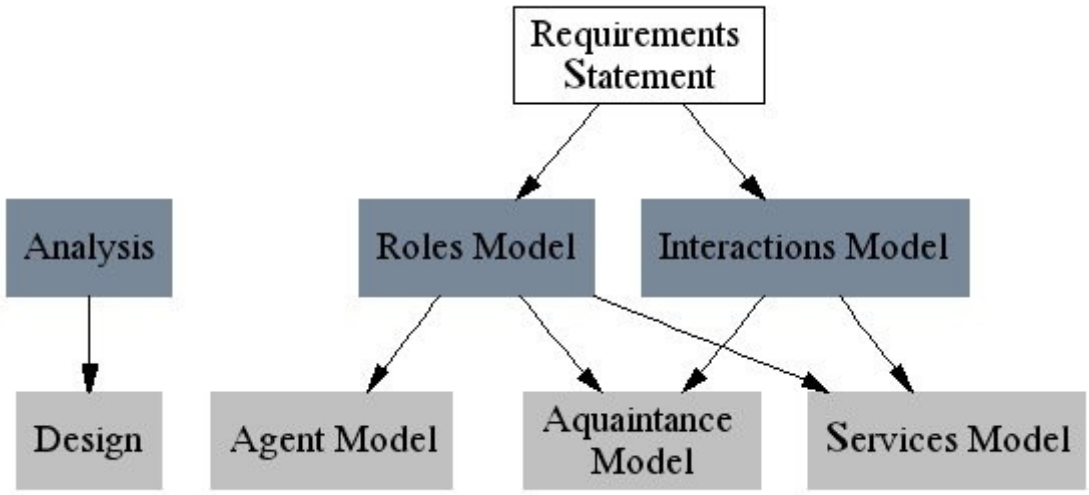

Figure 3-7: Gaia diagram

3.4.3.2.2 Analysis

In the analysis phase of Gaia the followings models must be created:

- Roles Model
- Interaction Model

In order to accomplish this, the following steps will be executed:

- Identify the roles in the system
- For each role identify and document the associated protocols
- Elaborate the roles model using the protocol model
- Iterate stages

3.4.3.2.2.1 Roles Model

The Roles Model identifies the key roles in the system.

It is composed by: Responsibilities and Permissions:

- Responsibilities: Determine the functionality of a role. There are to types of responsibilities:
    - Liveness responsibilities: Describe those states of affairs that an agent must bring about, given certain environmental conditions. They are expressed as: ROLENAME = expression. Where “expression” is compound of:
        - Activities: Activities carried by the agent without interaction with others roles.
        - Protocols: The way it can interact with others roles.
    - Safety responsibilities: Are invariants. They are expressed as predicates.
- Permissions: “rights” associated with a role
    - Resources that can be used.
    - Resource access permissions.
    - Permissions are: reads, changes and generates related to information, we can use also supplied.

3.4.3.2.2.2 Interaction Model

The Interaction Model is a set of all protocol definitions.

Each protocol definition is composed of:

- Purpose
- Initiator
- Responder
- Inputs
- Outputs
- Processing

3.4.3.2.3 Design

In the design phase of Gaia the followings models have to be created:

- Agent Model

- Services Model
- Acquaintance Model

In order to accomplish this, the following steps will be executed:

- Create agent model
- Develop services model
- Develop acquaintance model

3.4.3.2.3.1 Agent Model

The Agent Model identifies agent types. We can create an agent type tree in which each agent type can assume a set of agent roles.

3.4.3.2.3.2 Services Model

The Services Model identifies mains services required. Each service contains:

- Inputs: Derived from protocol model
- Outputs: Derived from protocol model
- Pre-conditions: Derived from safety properties
- Post-conditions: Derived from safety properties

3.4.3.2.3.3 Acquaintance Model

The Acquaintance Model documents lines of communication between agents. They can be represented by graphs with nodes and arcs.

3.4.3.2.4 Advantages

The Gaia methodology deals with a great variety of multi agents systems. It also deals at the macro-level, relations between agents, and micro-level, the agent itself.

3.4.3.2.5 Disadvantages

Gaia is of less value in open and unpredictable domain of Internet applications because it does not take the agent space into account; on the other hand it has been proven as a good approach for developing closed domain agent-systems.

3.4.3.3 SODA:

SODA [17] [16] (Societies in Open and Distributed Agent spaces) is a methodology for the analysis and design of Internet-based applications, in which Gaia was not of great value because SODA concentrates on the inter-agent issues.

SODA has two different phases: analysis and design and as in Gaia, in each phase, some models will be defined. Figure 3-4 show them.

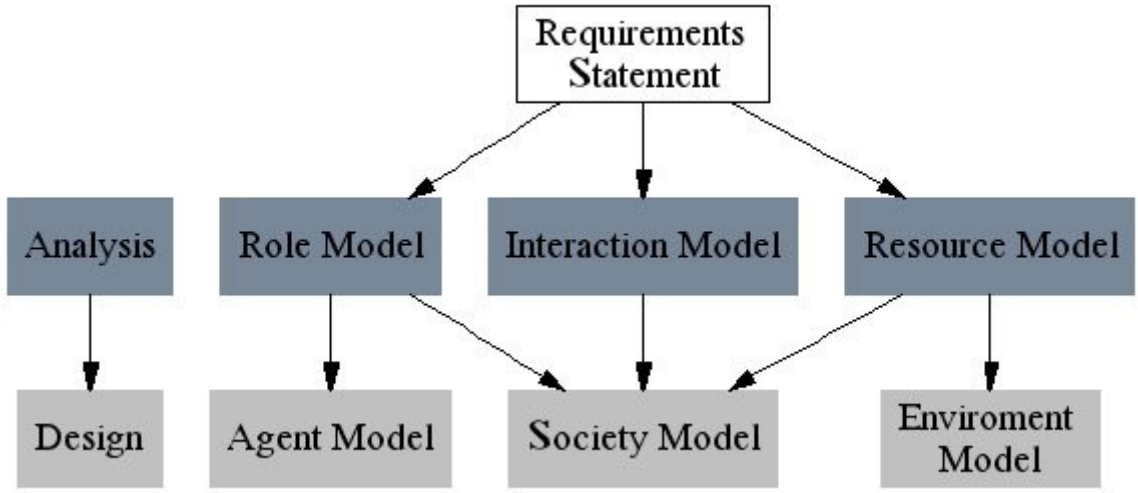

Figure 3-8: SODA diagram

3.4.3.3.1 Analysis:

The SODA analysis phase focuses on three distinct models:

The role model:

Application goals are modeled in terms of "tasks" to be achieved. They are expressed in terms of responsibilities, competences and resources. They are also classified as individual or social ones.

The resource model:

The application environment is modeled in terms of available services, which are associated to abstract resources.

The interaction model:

The interaction involving roles, groups of agents and resources is modeled in term of interaction protocols and interaction rules.

3.4.3.3.2 Design:

The SODA design phase focuses on three strictly related models:

The agent model:

Individual and social roles are mapped upon agent classes

The society model:

Groups of agents are mapped onto societies of agents, to be designed around coordination abstractions.

The environment model:

Resources are mapped onto infrastructure classes, and associated to topological abstractions.

SODA enables open societies to be designed around a suitably-designed coordination media, and social rules to be designed and enforced in terms of coordination rules.

SODA is the first methodology in take the agent space into account, so it provides specific abstractions and procedures for the design of agent infrastructures.

3.4.4 The OMT-AAII-MaSE Branch

Over the OMT OO methodology, the AAII agent oriented methodology is build taking special emphasis on the internal architecture of agents. MaSE derives from AAII improving it with more flexibility and improving the AAII deficiencies in organization structure.

3.4.4.1 OMT:

Object Modeling Technique (OMT) [18] involves System Analysis and Systems Design, is one of the precursors to the Unified Modeling Language (UML). There are three main diagrams in OMT which represent three different but complementary views of the system: Object, Dynamic, and Functional.

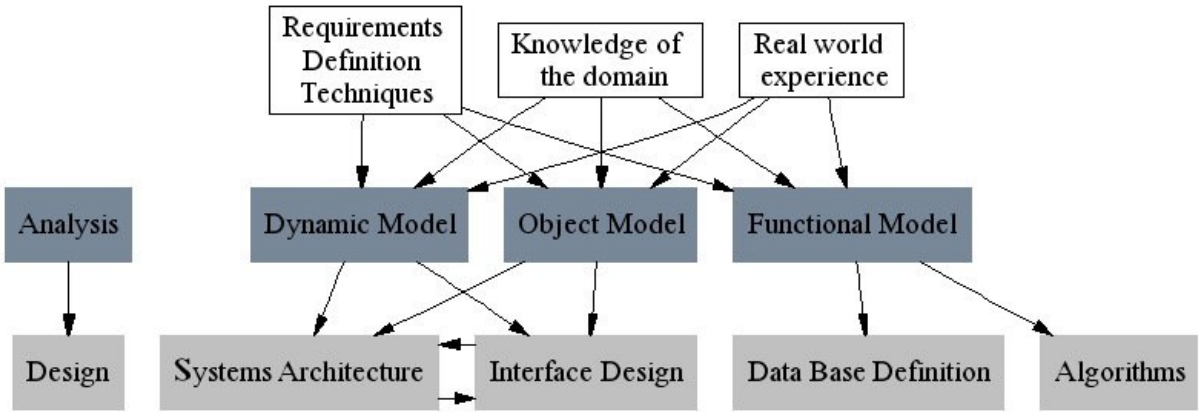

Figure 3-9: OMT diagram

Object Model:

It specifies the static structure of object and their relationships, as well as their attributes and methods. It represents the static structure of the system.

Dynamic model:

It specifies the aspect of a system which changes over time. It captures the essential behavior of the system.

Functional Model:

It specifies the data value transformations in a system. Describes what the system does but not how it is done.

Each phase of the process transforms some inputs to outputs, starting at a high level of abstraction and progressing to a more detailed level of abstraction that ultimately represents the problem solution. The more important items of the analysis and design phases are show in Figure 3-5.

3.4.4.2 AAII:

The AAII (Australian Artificial Intelligence Institute) [16] [19] methodology models a multi agent system at two levels of abstraction.

The External Viewpoint refers to the overall system that is decomposed into agents, "modeled as complex objects characterized by their purpose, their responsibilities, the services they perform, the information they require and maintain, and their external interactions". This viewpoint considers two models:

- The Agent Model

It captures the static organization of the agents. It specifies the types of agents that compose the system and the instances that will be present at run-time.

- The Interaction Model:

  It captures the dynamic organization of the agents.

The Internal Viewpoint defines elements required by individual agents to perform their actions. This viewpoint considers three models:

- Belief Model

  Beliefs are modeled as classes of concepts.

- Goal Model

  It defines the types of goals that an agent may have.

- Plan Model

  Plans are composed of nodes and transitions, with three different types of nodes: start nodes, internal nodes and end nodes.

3.4.4.3 MaSE:

MaSE [20] [16] has two phases: analysis and design. In each phase some items will be produced as show in Figure 3-6.

The objective of MaSE analysis is to produce a set of roles whose tasks describe what the system has to do to meet its overall requirements.

The MaSE analysis consists of three steps:

- **Capturing goals:**

  This step takes an initial system specification and transforms it into a structured set of system goals.

- **Applying uses cases:**

  This step captures a set of use cases from the initial system context and creates a set of Sequence Diagrams to help the system analyst identify an initial set of roles and communications paths within the system.

- **Refining roles:**

This step transforms the structured goals and Sequence Diagrams into roles and their associated tasks, generally one goal is transformed to a role.

The MaSE design process has four steps:

- **Creating agent classes**

  Agent classes are created from the roles defined in the analysis phase.

  An Agent Class Diagram is created.

- **Constructing conversations**

  This step defines the details of the inter-agent conversations.

- **Assembling Agent Classes**. Here the internals of the agents' classes are created.

  Agent Architecture is created.

- **System design**

  In a Deployment Diagram the numbers, type and location of agents will be shown.

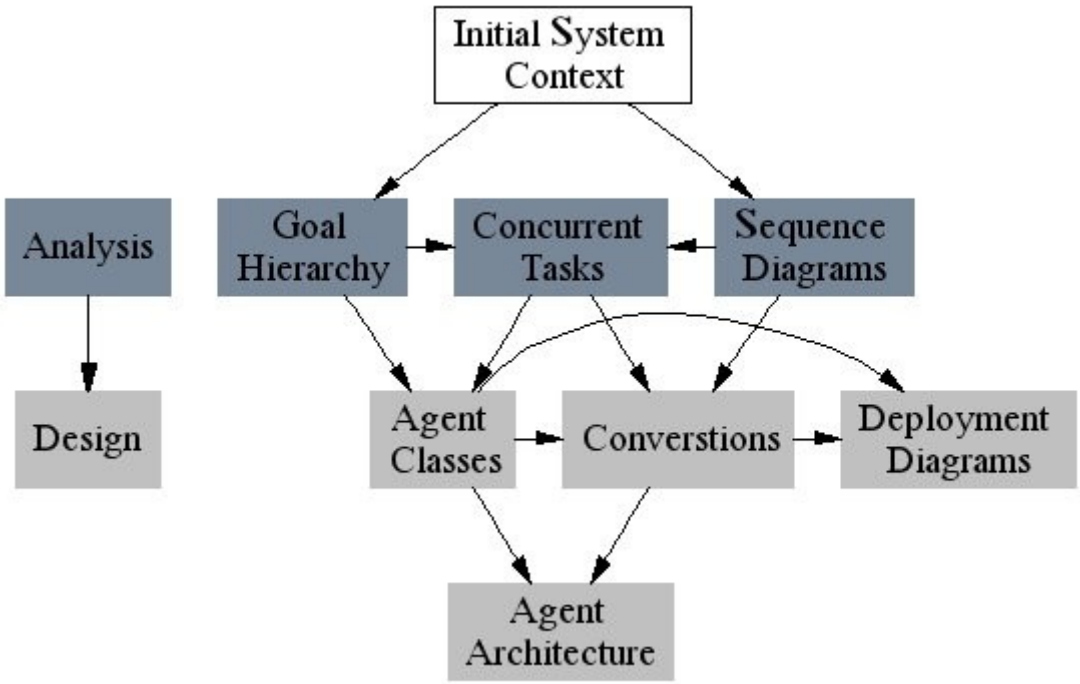

Figure 3-10: MaSE diagram

The MaSE methodology is comparable to the Gaia methodology in terms of defining roles the system. Within MaSE, however, a designer has a lot of freedom in not only defining the internal structure of the agents but also in defining the conversations between them. This methodology pays careful attention to the organization structure.

3.4.5 The OPEN-OPEN/Agents Branch

OPEN is a third generation OO methodology and OPEN/Agents extends it to adapt OPEN to deal with intelligent agents projects.

3.4.5.1 OPEN:

OPEN (Object-oriented Process, Environment and Notation) [21] [8] is essentially a framework for third generation object-oriented analysis.

OPEN extends the notion of a methodology by including a process model together with guidelines for constructing versions of this model tailored to the needs of individual organizations and problem types. The process model is an object model and, as such, is adaptable.

3.4.5.2 OPEN/Agent:

Because of the adaptability of OPEN, adding further support for the design of intelligent agents is feasible. Such potential extensions [8] have been a priori designed into the metamodel architecture of the OPEN Process Framework (OPF).

OPEN/Agents is still not a mature methodology, but OPEN will provide a high degree of flexibility to the user organization.

3.4.6 The RUP-MESSAGE and the RUP ADELFE Branches

The RUP OO methodology serves as a base for the MESSAGE and ADELFE agent oriented methodologies

3.4.6.1 RUP

RUP (Rational Unified Process) [22] provides a generic software engineering project lifecycle framework.

Some of the advantages of the RUP methodology are:

- It covers all the principal life cycle stages of software development.

- It can be applied to different application areas and different application sizes in different organizations.
- It takes into account the incremental and evolutionary nature of software development.

3.4.6.2 MESSAGE

MESSAGE (Methodology for Engineering Systems of Software Agents) [22] takes UML as a starting point and adds entity and relationship concepts required for agent-oriented modeling.

MESSAGE is based on the RUP OO methodology, but does not develop some of its phases like implementation and testing.

3.4.6.3 ADELFE

ADELFE [23] is based on object-oriented methodologies, follows the Rational Unified Process (RUP) and uses UML and AUML notations. It is not a general methodology, but concerns applications that require Adaptative Multi-Agent System design (AMAS).

3.4.7 Prometheus

The Prometheus methodology [24] [25] is a detailed one, which aims to cover all of the major activities required in the developing agent systems. The aim of Prometheus is to be usable by expert and non-expert users.

3.4.8 Cassiopeia

The Cassiopeia methodology [26] is a way to address a type of problem solving where collective behaviours are put into operation through a set of agents. It is not targeted at a specific type of application nor does it require a given architecture of agents. However, it is assumed that although the agents can have different aims the goal of the designer is to make them behave cooperatively. Cassiopeia relies on several concepts, namely those of role, agent, dependency, and group. The main idea is that we view an agent as nothing else but a set of roles.

3.4.9 Tropos

Tropos [27] is a novel agent-oriented software development methodology founded in two key features:

- The notions of agent, goal, plan and various other knowledge level concepts are fundamental primitives used uniformly throughout the software development process.

- A crucial role is assigned to requirements analysis and specification when the system-to-be is analyzed with respect to its intended environment.

3.4.10 MASSIVE

MASSIVE (MultiAgent SystemS Iterative View Engineering) [28] development method provides:

- A multiagent system specific product model that is used to describe the design and the implementation of the target system.
- A macro process model that covers the entire life cycle of the system development as well as several micro process models that are used to describe particular aspects of the target system.
- An institutional framework supports learning and reuse over project boundaries and allows the project management to configure the process and product models of a particular project.

3.5 UML and Java

Agent UML will be used for the modelling of the agents. This is because Agent UML is based on UML, a standard language for OO projects description. Agent UML makes some extensions to UML to allow it to deal with agents and agents' concepts.

Java will be used to implement the projects as it is a well suited language for dealing with agents. First of all, as agent oriented languages are only prototypes by now and not very used, an object oriented language is the first option to the development of agents. Then, among the object oriented languages like C++, Java, Python, Eiffel, CLOS, Smalltalk, Objective-C and Ruby, the most used and known nowadays are C++ and Java. These languages have compilers in almost every platform and operative system and most programmers feel comfortable with them. But Java is better suited to manage threads and concurrent programming in a general way and not depending on specific libraries as C++. So as agents will need intense use of threads and concurrent programming, Java will be adopted as the language for the implementation of the project.

3.6 Comparison of the Approaches

Here the most significant agent oriented methodologies are compared [16] [9]. The different phases that involve each methodology are enumerated. There is also a brief description of the

advantages and disadvantages of each methodology. At last the specific target of the methodology is mentioned. These characteristics may not be applicable to all the methodologies so some blanks will appear in the table.

	Phases	Advantages	Disadvantages	Target
AOR	Analysis Design	Uses UML	Only representation	Organizational information systems
PASSI	Analysis Design (Integrated)	Analysis and design from requirements implicit in creation of models		
MESSAGE	Analysis Design	Stepwise refinement in its analysis	Not well definition of the design process	
ADELFE	Requirements Analysis Design Implementation Test			Adaptative Multi-Agent System design
AAII	Analysis Design	Emphasizes internal agent architecture	Organizational structure sketchy	
MaSE	Analysis Design	Latitude for structure and communication in agents.		
Gaia	Analysis Design	Deals with macro and micro levels	Not well suited for Internet applications	Closed domain agent systems
SODA	Analysis Design	Take the agent space into account	Does not emphasize micro level	Internet-based applications
Prometheus	Analysis Design	Easy use		
Cassiopeia	Analysis Design			Cooperative architectures

Tropos	Requirements Design Implementation	Empathizes early requirements		

Table 3-1: Methodologies comparison

3.7 Conclusions

In this chapter the concepts of the OO paradigm have been exposed. Then, the agent concept has been explained and described what the agent oriented paradigm adds to the OO paradigm.

Then the main methodologies of agent oriented development have been explained and compared. Their differences make them suitable for different projects and different working organizations. Gaia is chosen for this project because it focuses both in micro and macro levels, for its clarity and its target to close domain agent systems.

At the end the adoption of Java for this project is explained.

4 Analysis and Design

4.1 Introduction

In this chapter the analysis and design of the project will be developed. At first, a description of the interface used to interact with the railway model will be provided in order to understand the possible available interactions. Then, the physical railway model is described because this will help in knowing the requirements.

Once the model and its possible interactions are described, the Gaia analysis is applied and the Roles and Protocol Models are obtained. Then, the Gaia design is applied and the Agent, Services and Acquaintance Models are obtained.

At last, the most important functionalities that involve several protocols are modeled using sequence diagrams in AgentUML.

4.2 IntelliBox Protocol

The railway model at the IAS is controlled by an Intellibox device. This device will send signals to all items in the model and will receive information from tracks. A serial port is available in the Intellibox to enable its control from an external device. A series of commands are available to be sent to this interface and control the model.

Here is a description of the commands that will be used in this project:

- Intellibox control commands:

  Go: Switch on the Intellibox.

  Stop: Switch off the Intellibox

- Train control commands:

  L {Lok#, [Speed], [FL], [Dir]}: This command allows to control the locomotives of the model. It needs at least the first argument which is the address of the Locomotive the command refers to. If only this argument is given, the intellibox will reply with the state of the locomotive. The rest of the arguments are optional and can be used in any combinations. For the purpose of this project, only the following will be used:

  - Speed: Speed of the locomotive. 0 = Inertial stop.

- FL: Is the state of the light of the locomotive. 0 = off, 1 = on.
- Dir: Is the direction of the locomotive. 1 or f = forward, 0 or r = reverse.

- Turnout control commands:

  T {Trnt#, [Color], [Status]}: This command allows controlling the turnouts of the models. A turnout is any accessory of the model like bifurcations, semaphores and lights. The first argument is the address of the turnout the command refers to. If only this argument is used, the Intellibox will respond with the state of the turnout. The rest of the arguments are optional and can be used in any combination:

  - Color: It refers to the state of the turnout. 0 or r = red, 1 or g = green. This makes sense in semaphores, but in bifurcations it only means the state in which the bifurcation is.
  - Status: This defines if the turnout is active or inactive. Status will be use always as active (= 1) in this project.

- Tracks commands:

  SS {mod}: This command allows to know the state of the tracks that correspond to the module specified in the argument. It only will say if a track is empty or busy.

4.3 The Railway Model

The railway model at the IAS will be used to the development of this project. The diagrams of the model are subdivided in two parts, the industrial level and the urban level. The industrial level is show in Figure 4-1, the urban level is show in Figure 4-3 and a legend with the symbols used in Figure 4-2:

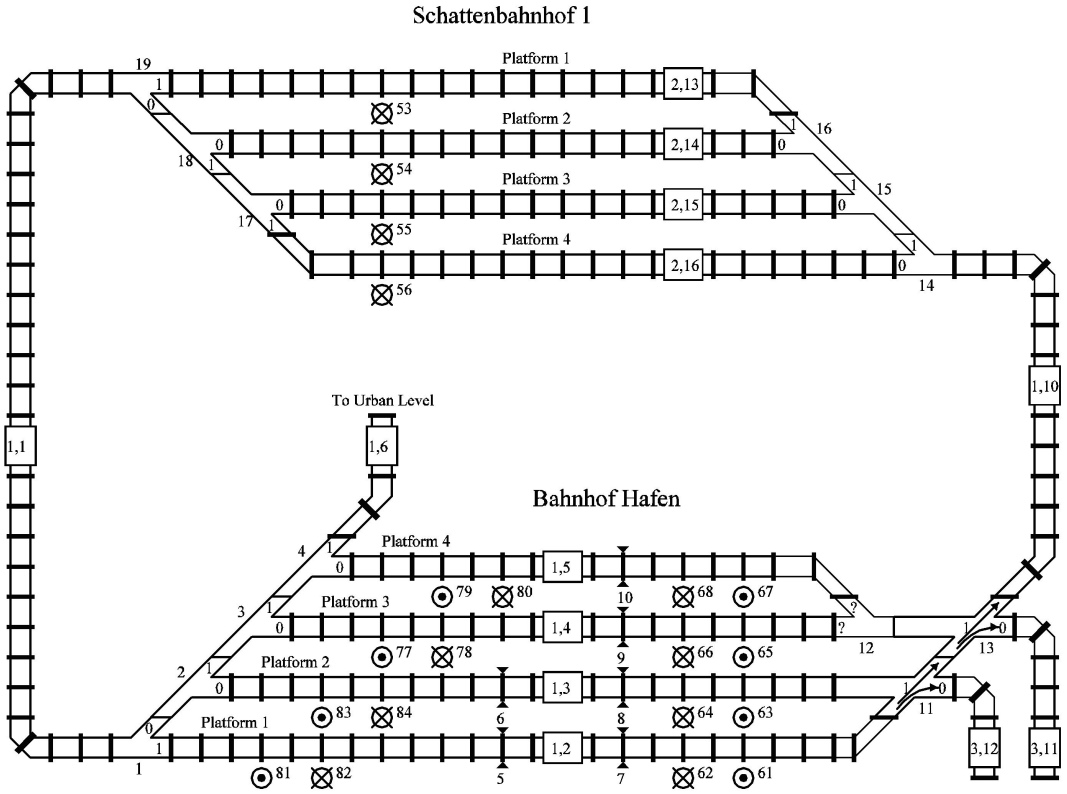

Figure 4-1: Industrial level

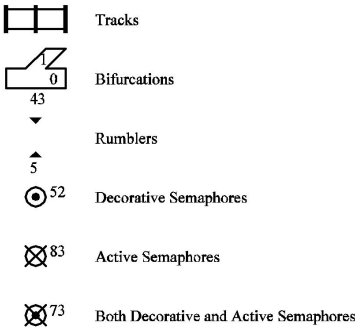

Figure 4-2: Legend

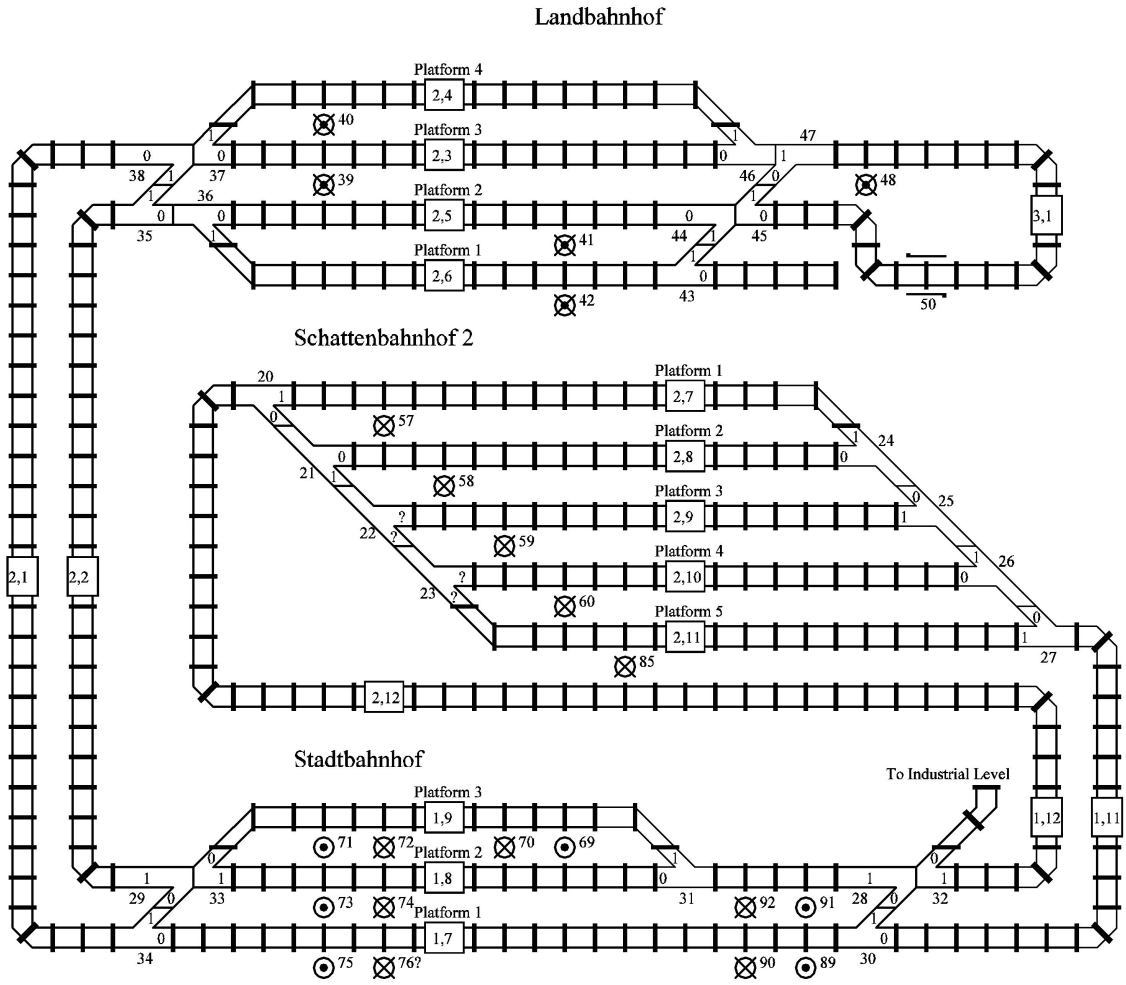

Figure 4-3: Urban level

The two levels are connected by a single track. The near bifurcations will be grouped and each track and group of bifurcation will receive a name.

There are two stations in the industrial level and three in the urban level. There are two evident paths in the model. The one in the industrial level goes from Schattenbahnhof1 to BahnhofHafen and then back to the first. The one in the urban level goes from Schattenbahnhof2 to Stadtbahnhof, then to Landbahnhof, then return to Landbahnhof passing by the loop track, then to Stadtbahnhof and at last to Schattenbahnhof2.

These paths will not allow communication between the different levels and the industrial path is very simple. Because that more routes for trains will be created. For example a train from the industrial level at certain times will go to the Landbahnhof station; the cleaner train will try to go to all tracks trying to not disturb the rest of the trains.

4.4 Analysis:

4.4.1 Identify the roles in the system:

At first, an enumeration of all items of the system will be done. Then they will be subdivided into resources and possible roles. After that, the possible roles will be refined identifying which ones could be resources of other roles. Finally the roles in the system will be identified.

4.4.1.1 Items Enumeration

An item classification can be done:

- **Active Items**: Items in the system that perform actions.
- **Passive Items**: Items in the system that do not perform any action.
- **Environmental Items**: Information about the environment that will be needed.
- **Artificial Items**: Items that do not exist in the reality but are conventions used in the system.

In our project, the railway model, the following items can be found:

Active Items:

- The IntelliBox.
- The Trains.
- The Tracks: they are considered active as they can send signals
- Active Semaphores: they are not seen in the model, but they will stop a train.

Passive Items:

- Bifurcations: they may be considered active as they perform action like change a track; but I will consider them as a change in their state and not like an action.
- Decorative Semaphores: the will show a green or red light only.
- Wagons.

Environmental Items:

- Time.
- Timetable: it will say at which time a train must be in one track or station.

Artificial Items:

- Station: It is a group of parallel tracks, it is considered as in the timetable a station can be selected as a destination instead of a single track.

4.4.1.2 Subdivision in Possible Roles and Resources

Now, to extract the roles from it the system requirements or functions will also be enumerated:

- A train must move in a safe way. No more than one train is allowed in the same track.
- A train will try to follow its timetable and change it if it is not possible to follow it.
- The paths from one location in the model to another location must be found.
- A train must be able to go from any place in the model to a specific track.
- When a train goes to a station, it will go to an empty track and stop.

With all this, it is possible to identify the possible roles. Items will be divided into resources and possible roles:

- **The IntelliBox:** It will only receive commands and forward them to the model, and send feedback from the model. It can not take any function on the system, so the IntelliBox will be a **resource**.
- **The Trains:** They will have to go from one place to other in a safe way, and follow its timetable. They can accomplish some functions of the system so they are a **possible role**.
- **The Tracks:** They will inform if a train is on them or not, but they may help finding a path for the trains, so they are a **possible role**.
- **Active Semaphores:** They will be able to stop a train in a track, they can help in the function of moving a train in a safe way and to stop at a station, they will be a **possible role**.
- **Bifurcation:** They can help in taking one train from one place to a specific place, so they are a **possible role**.

- **Decorative Semaphores:** They are only decorative and do not help in any function of the system, so they are a **resource**.
- **Wagons:** They will be carried by trains and do not perform any of the functions of the system, they are a **resource**.
- **Time:** It is only consulted by other roles, it does not perform any function of the system, so it is a **resource**.
- **Timetable:** It provides information about some functions and can be modified, but does not realize any function, so it is a **resource**.
- **Stations:** They are abstractions, but they can choose a track for an incoming train to the station so they are a **possible role**.

So, in consequence:

Possible roles: Trains, Tracks, Active Semaphores, Bifurcations, Stations.

Resources: IntelliBox, Decorative Semaphores, Wagons, Time, Timetable

4.4.1.3 Identifying the Roles

Now, in order to define the final roles, it will be necessary to define if some of the possible roles can be resources of others roles without loosing functionality and without overloading any role.

- **Trains:** They may be considered as resources for the tracks and tracks will control the trains, but this approach will be not as natural as consider trains as independents roles, so I will consider trains as a **role**.
- **Tracks:** They can be considered as resources of trains, but then all the path search functionality will be in the trains and it can be excessive that a role has almost all the functions of the system. Also tracks can follow the position of each train and this will help some functions of the system, so tracks will be considered a **role**.
- **Active Semaphores:** They can be considered as resources of the tracks and instead of overloading the tracks functions; they will help to stop a train in that track or stopping a train to go to one track which is busy. So the Active Semaphores will be considered as **resources for the tracks**.
- **Bifurcations:** Maybe this will be one of the most difficult possible roles to define as a role or as a resource. They can be considered as resources of tracks; tracks will change their state to send the train in a secure way. The state of a bifurcation is the position of its corresponding elements in the railway model which will direct trains in different

directions. Also they can be considered as specials tracks which can change its state to take the train to different tracks. In favor to be considered resources is that a whole train cannot stop in a track because it is too small, tracks do not send signals if a train is on them and that tracks may take an important role in the path search and safety functionality. On the other hand, they are difficult to be a resource of a specific track but of several tracks and some conflicts could arise. So the simple way may be consider bifurcations as specials tracks with a bit different functionality and they will be considered a **rol**e.

- **Stations:** They are not obviously resources of any role and the will have a defined functionality choosing the track a train must stop at, so they will be considered a **role**. Another possible approach could be to consider the tracks of a station as resources of the station, but then there will be loss of generality in tracks.

So, in conclusion the roles with the requirements that they will take in the system will be the following; some requirements will correspond to different roles as they will work together to accomplish them:

- Trains
    - A train must move in a safe way. No more than one train is allowed in the same track.
    - A train will try to follow its timetable and change it if it is not possible to follow it.
    - A train must be able to go from any place to a specific place.
- Tracks
    - A train must move in a safe way. No more than one train is allowed in the same track.
    - The paths from one location in the model to another location must be found.
    - A train must be able to go from any place to a specific place.
- Bifurcations
    - A train must move in a safe way. No more than one train is allowed in the same track.
    - The paths from one location in the model to another location must be found.

  - A train must be able to go from any place to a specific place.

- Stations
  - When a train goes to a station, it will go to an empty path and stop

4.4.2 Identify the protocols for each role

The movement of the trains will consider three different factors:

- Safety: guarantees the security in movements of trains.
- Path: enables to find paths from an origin track to a destiny track.
- Preference: will determine preferences for trains in tracks and bifurcation avoiding trains with less preference to go to this track or bifurcation.

They are described here:

- **Safety**: for safety a reserving track method will be used. A train will be able to move to a track only if it has been reserved for this train before. The reserving method will work in the following way: when a train intends go to a track, first the train will send a reservation request to the next track. If the request does not succeed, then if the train is moving, it will stop and will wait until the track is free or will search other path to the destiny (Figure 4-4). If the request succeeds, the reserved track will try to reserve the next track. If this second request does not succeed, the first track will indicate that the train will have to stop in this track (Figure 4-5). If the second request is successful, then the track is reserved and the train can move to this track and repeat the same process (Figure 4-6).

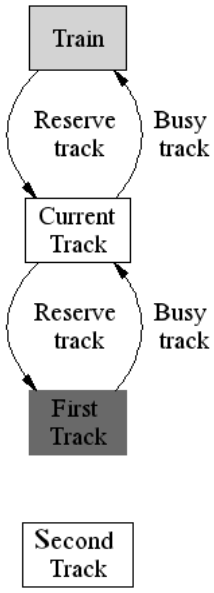

Figure 4-4: safety method (a)

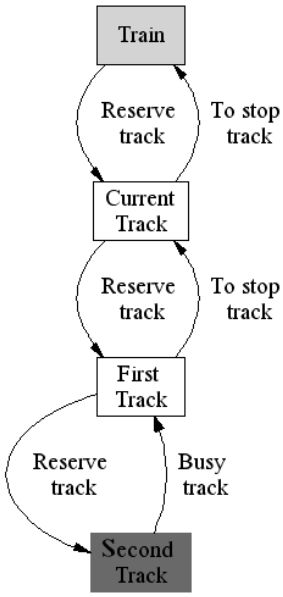

Figure 4-5: safety method (b)

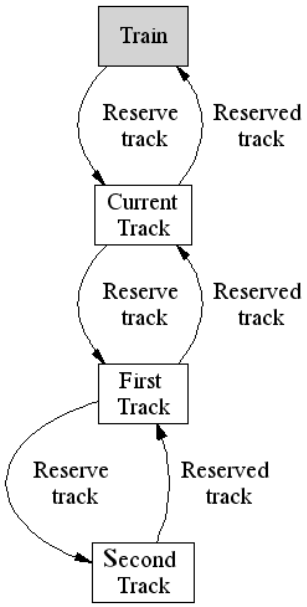

Figure 4-6: safety method (c)

If a bifurcation is between two tracks, the bifurcation has also to be reserved. If it does not succeed it will be the same as if the next track were busy (Figure 4-7 and Figure 4-8). If it succeeds, it will be the same as if the bifurcation did not exist (Figure 4-9, Figure 4-10 and Figure 4-11). This is because the impossibility of detecting trains in a bifurcation and to stop trains on them.

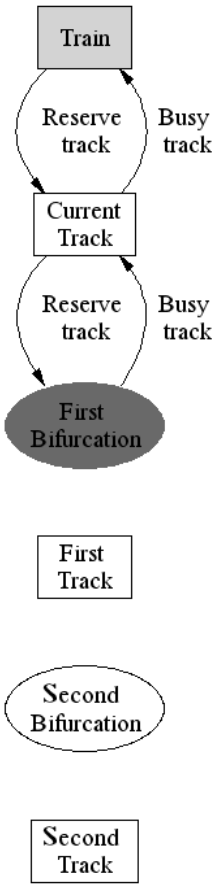

Figure 4-7: safety method (d)

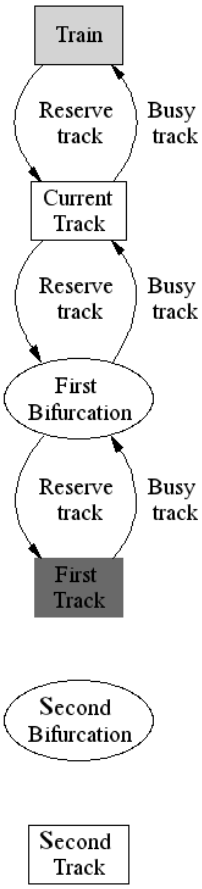

Figure 4-8: safety method (e)

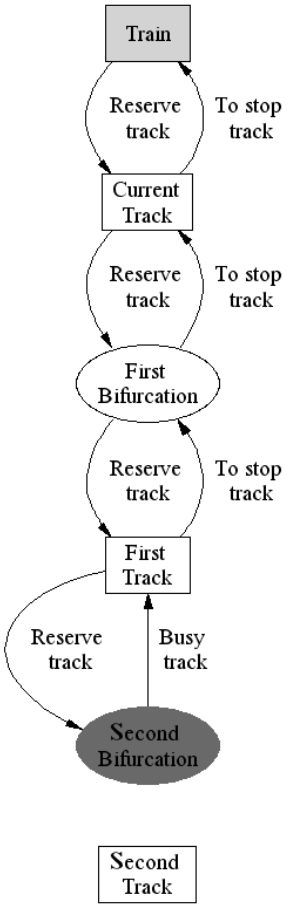

Figure 4-9: safety method (f)

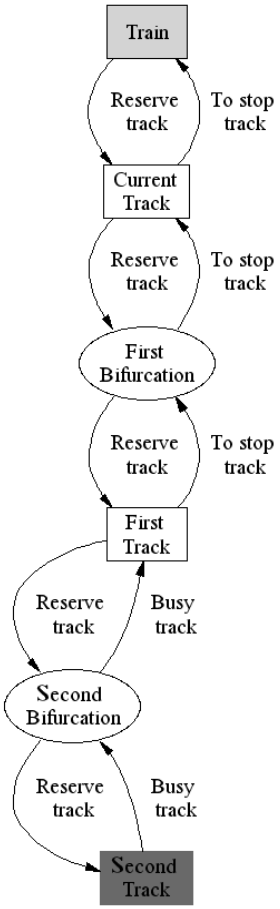

Figure 4-10: safety method (g)

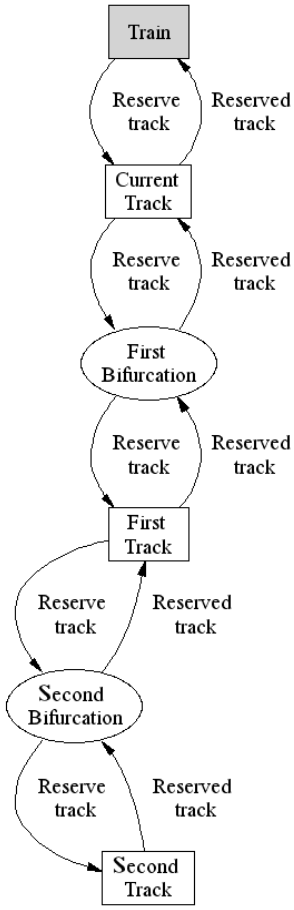

Figure 4-11: safety method (h)

This method of reserving two tracks is necessary in this model because if the train only reserves the next track, due to delays in the transmission it is possible that the train does not have time to stop if next track is busy. With this method the train will know if a track is busy and if it must stop in the previous one, even before arriving to that previous track.

It is also important to mention that this method will be used both in forward and backward directions.

At last, when a track detects that a train has left the track, it must release itself (allowing further reservations) and the near bifurcation if it exists.

- **Path**: A train will be able to ask for paths for a certain destination. A train will only move towards tracks of one of the found paths to the destination. A spreading method will be used to find paths. This method will work like this; when a train looks for a path, it will ask its current track, the current track will forward the request to both sides of the track and add its name to the route (Figure 4-12).

When a track receives a path request from a track, it will check if the name of the track is in the path, if not it will forward the request to the opposite side of the track (Figure 4-13), otherwise it will ignore the request. This avoids paths that pass twice over the same track. If the request comes from a bifurcation, the track will check again for its name in the path and if it is not there, the track will forward the request to the opposite side of the track and to the bifurcation again (Figure 4-14). This is because paths may pass more than once over a bifurcation but with the bifurcation in different states.

When a bifurcation receives a request, it will forward the request to the possible tracks that a train may take from the track that send the request (Figure 4-15).

When a track receives a request and the track is the destiny, the track will respond the initial asker of the request with the whole path (Figure 4-16).

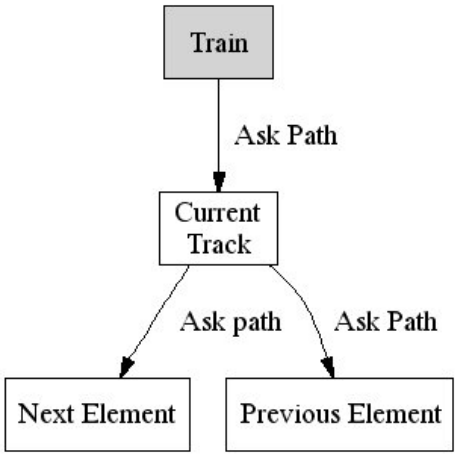

Figure 4-12: find path method (a)

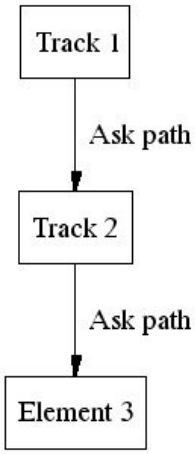

Figure 4-13: find path method (b)

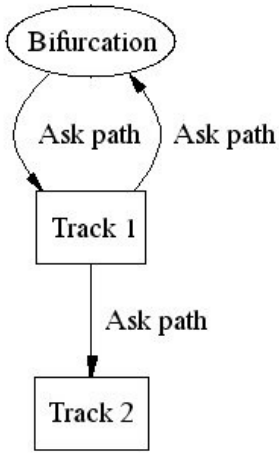

Figure 4-14: find path method (c)

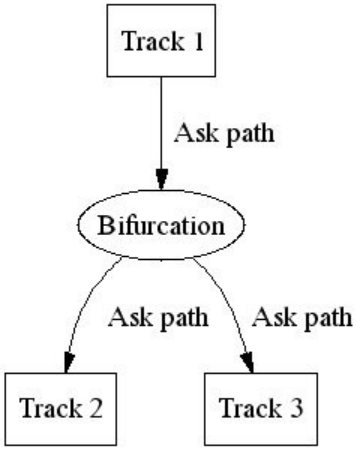

Figure 4-15: find path method (d)

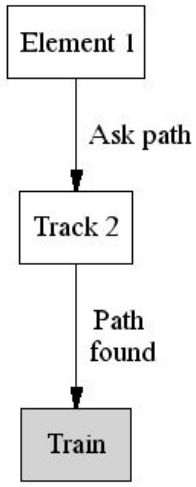

Figure 4-16: find path method (e)

- **Preference**: A train will be able to move towards a track only if there is preference in the direction the train try to go. The preference will be a number that determines the priority of a train over a track in a direction. In order to determine preference, the following method will be used:

  When a train decides to follow a path, it will send an intention message that will be forward over the path. Depending on the distance of the train to the track, the standard direction of the track, the delay of the train over its timetable and the intentions of other trains in the same track, an intention number will be saved in each track and bifurcation.

  A train can also ask the preference of a track, which is a number representing the preference of the asking train in this track computed with the actual intentions saved in the track and information from the asking train, to be able to select a path with the most preference possible.

  Preferences also will be modified when a train leaves a track.

Apart from this, a train must be able to know which track is in front of it and which one is behind of it to be able to follow a path, so a protocol will be need for this.

Also, a train must know if there is a semaphore in the track it intends to stop and ask the track to close it if available. These factors can be implemented in different ways, but this description is enough for defining the protocols that will be used.

Trains protocols:

- TrainReserveTrack: Ask the current track to reserve the next track in the current path.

- AskPathTo: Ask the current track for paths to a destination.
- AskPreference: Ask a track for the preference.
- TrainSendIntentions: Send the intentions of the movement of the train along the path.
- AskTrack: Ask a track for a Station.
- AskDirection: Ask the current track for the direction of the train over the track; return next and previous track respect to the train.
- AskStop: Ask a track to close its semaphores.
- AskOpen: Ask a track to open its semaphores.

Tracks protocols:

- TellArrivedTrack: This will tell a train that it is above the sending track.
- TellReleasedTrack: This will tell a train that it has leave the track.
- TellDirection: Will send to the train information of the next and previous track respect to the train.
- TrackReserveTrack: Will reserve a track because a petition from a train.
- TrackState: Will send the status of the track.
- TrackAskPath: Will ask a path to a destiny track.
- TellTrackPreference: Send the preference of this track
- TrackSendIntentions: Send the intentions of movement of a train along its supposed path.
- ReturnPath: Return the path to this track.
- ReleaseBifurcation: Will release a path of a bifurcation.
- StopState: Tells the train if the track has closed its semaphores.
- OpenState: Tells the train if the track has opened its semaphores.

Bifurcations protocols:

- BifurcationReserveTrack: Will reserve a track because a petition of a track.

- BifurcationState: Will send the status of the bifurcation in a specify direction.
- BifurcationAskPath: Will ask a path to a destiny track.
- TellBifurcationPreference: Will say the bifurcation preference in a path.
- BifurcationSendIntentions: Will send the intention of movement of a train along its supposed path.

Stations protocols:

- ReturnTrack: Will send the train a track that it must use to arrive to the station.
- AskPath: Ask a path. This may be used for the station to decide a suitable track for a train.
- AskPreference:
- GoTrain: Tell a train stopped in a station to leave the station.

With these brief protocols definitions, there is possible now to create the roles model.

4.4.3 The Roles Model

The Roles Model identifies the key roles in the system; they can be viewed as an abstract description of an entity's expected function. In the Roles Model, roles are described with its protocols, activities, permissions and liveness and safety responsibilities.

Role Schema: TRAIN

Description:

Try to follow a timetable with minimum delays, in a safe way and will not block others trains.

Protocols and Activities:

TrainReserveTrack, AskPathTo, AskPreference, TrainSendIntentions, AskTrack, AskDirection, AskStop, AskOpen, MoveTrain, EvaluatePreferences, AskNextStation, AskTime, WaitArrivedTrack.

Permissions:

reads	supplied *currentTrack*	*// the track the train is above*
	destinyStation	*// next destiny station for the train*
	supplied *destinyTrack*	*// track to be used in a station*
	supplied *possiblePaths*	*// path to a destiny*
	supplied *preference*	*// preference of a track*
	supplied *direction*	*// direction of the train over the track*
	supplied *stopMethod*	*// stop method for next track*
	currentTime	*// time of the system*
changes	*timetable*	*// timetable for the train*
	nextTrackState	*// state of next track*
generates	*currentPath*	*// the selected path at the moment*
	intentions	*// intended path*

Responsibilities:

Liveness:

MOVENEXTDESTINY = (AskTime·AskNextStation·GoNextStation)$^{\omega}$
GONEXTSTATION = (AskTrack·SelectPath·GoAlongPath)
SELECTPATH = (AskPathTo·EvaluatePaths·TrainSendIntentions)
GOALONGPATH = (ReserveTrack·ReactionToTrackStatus·WaitArrivedTrack)+
REACTIONTOTRACKSTATUS = (StopTrain* · SelectPath* · GoTrain*)
STOPTRAIN = (AskStop·MoveTrain)
GOTRAIN = (AskDirection·AskOpen·MoveTrain)
EVALUATEPATHS = ((AskPreference+)·AskTime.EvaluatePreferences)

Safety:

- nextTrackState = reserved

Table 4-1: Train role model

Role Schema: TRACK

Description:

Will follow the trains' position. Will manage the reserving, path finder and preference methods. They also will provide trains with information about the train direction and the semaphores of the track.

Protocols and Activities:

TellArrivedTrack, TellRealeasedTrack, TellDirection, TrackReserveTrack, TrackState, TrackAskPath, TellTrackPreference, TrackSendIntentions, ReturnPath, ReleaseBifurcation, StopState, OpenState, CloseSemaphore, OpenSemaphore, CalculatePreference, ProcessTrackStateChanges, ModifyState.

Permissions:

reads	supplied *trainIntentions*	*// next destiny station for the train*
	supplied *currentPath*	*// current path of the train that reserves the track*
	supplied *possibleTracks*	*// possible tracks of next bifurcation*
	trackSensor	*// information from the model about free or busy track*
changes	*trackState*	*// State of the track*
	supplied *nextTrackState*	*// state of next track*
	supplied *foundPath*	*// fragment of path found*
generates	*preference*	*// path to a destiny*
	trainDirection	*// direction of the train over the track*
	stopMethod	*// preference of a track*
	currentIntentions	*// track to be used in a station*
	openMethod	*// direction of the train over the track*

Responsibilities:

Liveness:

RESERVETRACK = (ModifyState·(TrackReserveTrack)*)

FINDPATH = ((TrackAskPath+) | ReturnPath)*

SPREADINTENTIONS = (TrackSendIntentions)

REPLYPREFERENCE = (CalculatePreference·TellTrackPreference)

FOLLOWTRAINS = (ProcessTrackStateChanges·ModifyState· (TellArrivedTrain| TellReleasedTrain)·(ReleaseBifurcation))ω

REPLYDIRECTION = (TellDirection)

STOPSTATE = ((CloseSemaphore)*·StopState)

OPENSTATE = ((OpenSemaphore)*·OpenState)

Safety:

- **True**

Table 4-2: Track role model

Role Schema: BIFURCATION

Description:

Will manage the reserving, path finder and preference methods. They also will provide trains with information about preference.

Protocols and Activities:

BifurcationReserveTrack, BifurcationStatus, BifurcationAskPath, TellBifurcationPreference, BifurcationSendIntentions, SetPath, CalculatePreference, ModifyState.

Permissions:

Reads	supplied *trainIntentions*	*// next destiny station for the train*
	supplied *currentPath*	*// current path of the train that reserves the track*
changes	*bifurcationState*	*// State of the track*
	supplied *nextTrackState*	*// state of next track*
	supplied *foundPath*	*// fragment of path found*
generates	*preference*	*// path to a destiny*
	currentIntentions	*// track to be used in a station*
	possibleTracks	*// possible tracks of this bifurcation from a specific track*

Responsibilities:

Liveness:

RESERVEBIFURCATION = (ModifyState·SetPath·(BifurcationReserveTrack)*)

FINDPATH = (BifurcationAskPath+)

SPREADINTENTIONS = (BifurcationSendIntentions)

REPLYPREFERENCE = (CalculatePreference·TellBifurcationPreference)

RELEASEBIFURCATION = (ModifyState)

Safety:

- **True**

Table 4-3: Bifurcation role model

Role Schema: STATION		
Description: This role will assign tracks to trains that intend to reach a station.		
Protocols and Activities: ReturnTrack, AskPath, AskPreference, AskTime, BuildPreferences.		
Permissions:		
reads	*Timetable*	*// timetable for the train*
	stationTracks	*// the track that are in the station*
	supplied *possiblePaths*	*// path to a destiny*
	supplied *preference*	*// preference of a track*
	currentTime	*// time of the system*
generates	*destinyTrack*	*// track to be used in a station*
	trackPreferences	*// preferences of the tracks of the station*
Responsibilities: **Liveness:** ASSIGNTRACK = ((AskPath)*·(AskPreference)*·(AskTime)*·(BuildPreferences)*·(ReturnTrack)) **Safety:** • **True**		

Table 4-4: Station role model

4.4.4 Interaction Model

Interactions between the various roles in the system are captured and represented in this model. In this model each inter-role interaction is defined. The attention is focused on the essential nature and purpose of the interaction, rather than on the precise ordering of particular message exchanges.

4.4.4.1 Protocols whose initiator is the Train.

TrainReserveTrack		**currentPath** **currentTrack**
Train	**Track**	
Ask the current track to reserve the next track in the current path		**nextTrackState**

Table 4-5: TrainReserveTrack protocol

AskPathTo		destinyTrack currentTrack
Train	Track	
Ask the current track for paths to a destination		possiblePaths

Table 4-6: AskPathTo protocol

AskPreference		possiblePaths
Train	Track Bifurcation	
Ask a track for its preference		preference

Table 4-7: AskPreference protocol

TrainSendIntentions		timetable currentPath currentTrack
Train	Track	
Send the intentions of the movement of the train along the path.		trainIntentions

Table 4-8: TrainSendIntentions protocol

AskTrack		destinyStation
Train	Station	
Ask the station for an arrival track		destinyTrack

Table 4-9: AskTrack protocol

AskDirection		currentTrack
Train	Track	
Ask the current track for the direction of the train over the track		trainDirection

Table 4-10: AskDirection protocol

AskStop		currentPath
Train	Track	
Ask a track to close its semaphores		stopMethod

Table 4-11: AskStop protocol

AskOpen		currentPath
Train	Track	
Ask a track to open its semaphores		openMethod

Table 4-12: AskOpen protocol

4.4.4.2 Protocols whose initiator is the Track

TellArrivedTrack		trackState
Track	Train	
Communicate a train that it is above the sending track		currentTrack

Table 4-13: TellArrivedTrack protocol

TellReleasedTrack		trackState
Track	Train	
Communicate a train that it has leave the track		currentTrack

Table 4-14: TellReleasedTrack protocol

TellDirection		trackState
Track	Train	
Will send a train information of the next and previous tracks respect to the train		trainDirection

Table 4-15: TellDirection protocol

TrackReserveTrack		currentPath
Track	Track Bifurcation	
Will reserve a track because a petition from a train		nextTrackState

Table 4-16: TrackReserveTrack protocol

TrackState		
Track	Track Bifurcation Train	
		trackState

Table 4-17: TrackState protocol

TrackAskPath		foundPath
Track	Track Bifurcation	
Will ask a path to a destiny track.		foundPath

Table 4-18: TrackAskPath protocol

TellTrackPreference		currentIntentions
Track	Train	
This will send the preferences of the track		preference

Table 4-19: TellTrackPreference protocol

TrackSendIntentions		currentPath
Track	Track Bifurcation	
This will send the intentions of movement of a train along its supposed path.		trainIntentions

Table 4-20: TrackSendIntentions protocol

ReturnPath		
Track	Train	
Returns the path to this track		foundPath

Table 4-21: ReturnPath protocol

ReleaseBifurcation		trackStatus
Track	Bifurcation	
This will release a path of a bifurcation		releasePath

Table 4-22: ReleaseBifurcation protocol

StopState		trackState semaphores
Track	Train	
This will communicate a train if the track has closed its semaphores for it.		stopMethod

Table 4-23: StopState protocol

OpenState		trackState semaphores
Track	Train	
This will communicate a train if the track has opened its semaphores for it.		openMethod

Table 4-24: OpenState protocol

4.4.4.3 Protocols whose initiator is the Bifurcation

BifurcationReserveTrack		currentPath
Bifurcation	Track	
This will reserve a track because a petition of a track		nextTrackState

Table 4-25: BifurcationReserveTrack protocol

BifurcationState		
Bifurcation	Track	
This will send the status of the bifurcation for a path.		bifurcationState

Table 4-26: BifurcationState protocol

BifurcationAskPath		foundPath possibleTracks
Bifurcation	Track	
Will ask a path to a destiny track.		foundPath

Table 4-27: BifurcationAskPath protocol

TellBifurcationPreference		currentPath
Bifurcation	Train	
This will send a train the bifurcation preference in a path.		Preference

Table 4-28: TellBifurcationPreferences protocol

BifurcationSendIntentions		currentPath
Bifurcation	Track	
This will send the intentions of movement of a train along its supposed path.		trainIntentions

Table 4-29: BifurcationSendIntentions protocol

4.4.4.4 Protocols whose initiator is the Station

ReturnTrack		trackPreferences stationTracks
Station	Train	
This will send a train the track it must use to arrive to the station.		destinyTrack

Table 4-30: ReturnTrack protocol

AskPath		
Station	**Track**	
Ask paths. This may be used by the station to decide a suitable track for a train.		**possiblePaths**

Table 4-31: AskPath protocol

AskPreference		possiblePaths
Station	**Track**	
Ask preference of tracks. This may be used by the station to decide a suitable track for a train.		**preference**

Table 4-32: AskPreference protocol

4.5 Design

4.5.1 Agent Model

This model will document the various agent types that will be used in the system under development, and the agent instances that will realise these agent types at run-time. Every agent will be composed of at least one role; this documentation is also provided in this model.

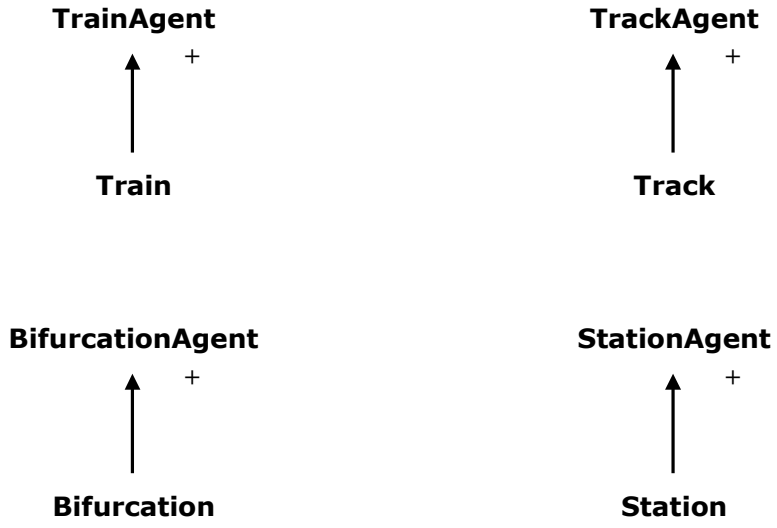

Figure 4-17: Agent model

4.5.2 Services Model

This model identifies the services associated with each agent role and specifies the main properties of these services. The services are derived from the list of protocols, activities, responsibilities and liveness properties of each role.

Service	Inputs	Outputs	Pre-condition	Post-condition
ask next station		*destinyStation*	**true**	**true**
ask next track	*destinyStation*	*destinyTrack*	**true**	**true**
search path	*destinyTrack*	*possiblePaths*	**true**	**true**
evaluate preferences	*possiblePaths preference direction currentTime*	*currentPath*	**true**	**true**
send intentions	*currentPath*	*intention*	**true**	**true**
reserve track	*currentPath*	*trackState*	**true**	**true**
move train	*currentPath direction*		*nextTrackState = reserved preference > 0*	**true**
ask time		*currentTime*	**true**	**true**
wait arrived track		*currentTrack*	*currentTrack ≠ nextTrack*	*currentTrack = nextTrack*
close semaphore	*currentPath*	*stopMethod*	exists semaphores in the correct direction	*semaphores = closed*
open semaphore	*currentPath*	*openMethod*	exists semaphores in the correct direction	*semaphores = open*
calculate preferences	*currentIntentions*	*preference*	**true**	**true**
process track state changes	*trackState currentPath*	*currentTrack*	*trackState = reserved*	*trackState = inUse*
modify state of track	*reservePathPetition currentPath currentTrack or trackSensor*	*trackState*	**true**	**true**
set path in bifurcation	*reservePathPetition currentPath currentTrack*	*bifurcationStat e*	*possibleTracks contains fragment of currentPath*	*bifurcationState are set to currentPath*
build preferences	*stationTracks possiblePaths preference*	*trackPreferenc e*	**true**	**true**

Table 4-33: Services model

4.5.3 Acquaintance Model

This model simply defines the communication links that exist between agent types. This may identify any potential communication bottlenecks.

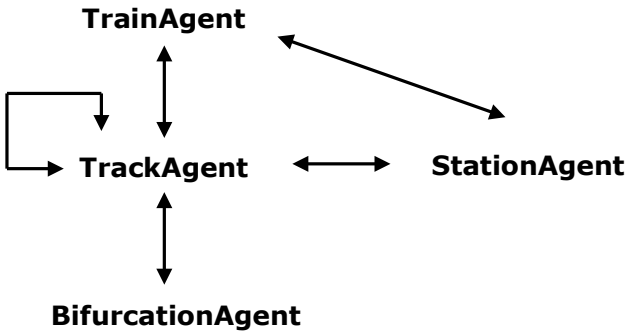

Figure 4-18: Acquaintance model

4.6 AgentUML Modeling

AgentUML is an extension to the standard UML language to allow it more flexibility dealing with agents systems. It is used here in order to represent the protocol interactions in the system in a dynamic way because it is not represented in any Gaia models.

First, the **reserve track** method will be modeled.

There are several cases:

- Cases without bifurcations between tracks
    - The first track could not be reserved

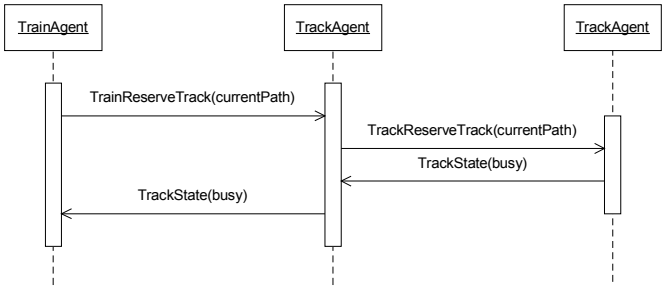

Figure 4-19: sequence diagram of the reserve track method

- The second track could not be reserved

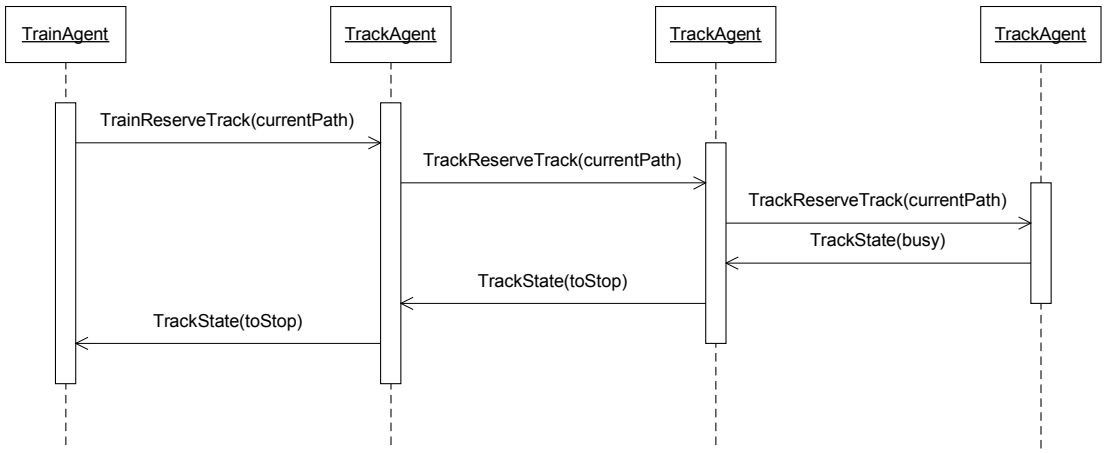

Figure 4-20: sequence diagram of the reserve track method

- The tracks could be reserved

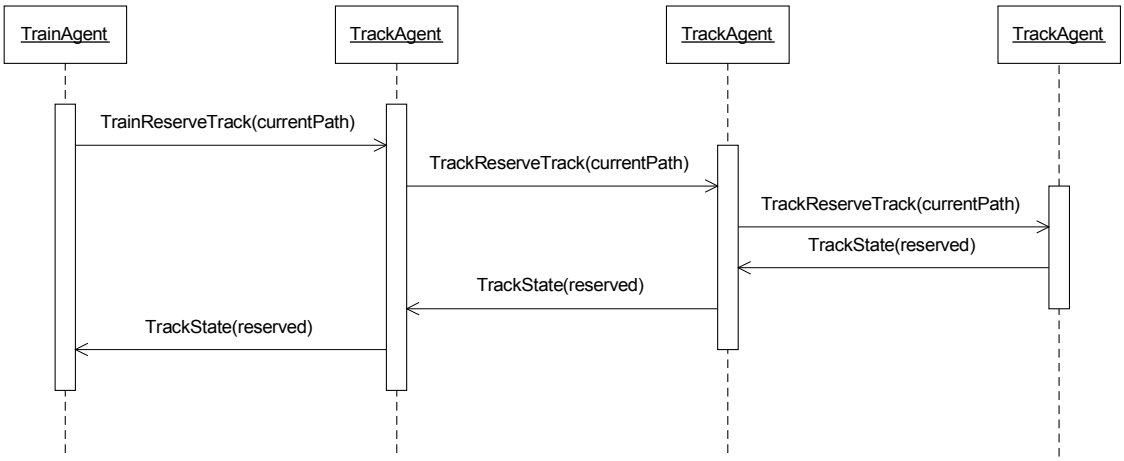

Figure 4-21: sequence diagram of the reserve track method

- Cases with bifurcations between tracks
    - The first bifurcation could not be reserved

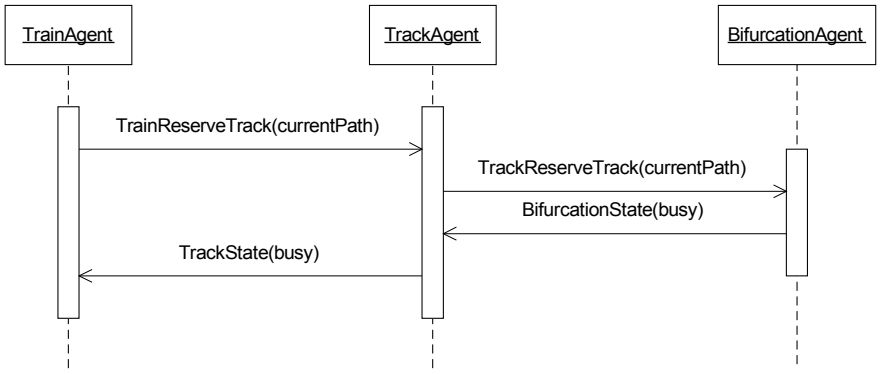

Figure 4-22: sequence diagram of the reserve track method

- The first tracks could not be reserved

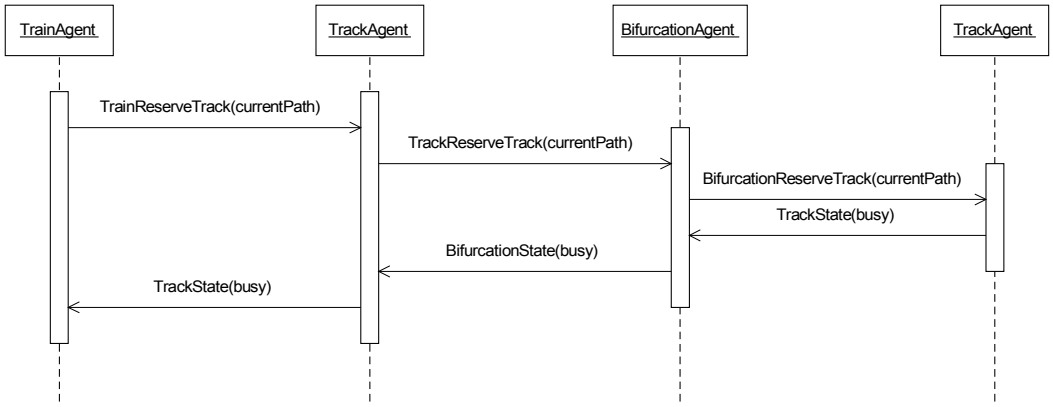

Figure 4-23: sequence diagram of the reserve track method

- The second bifurcation could not be reserved

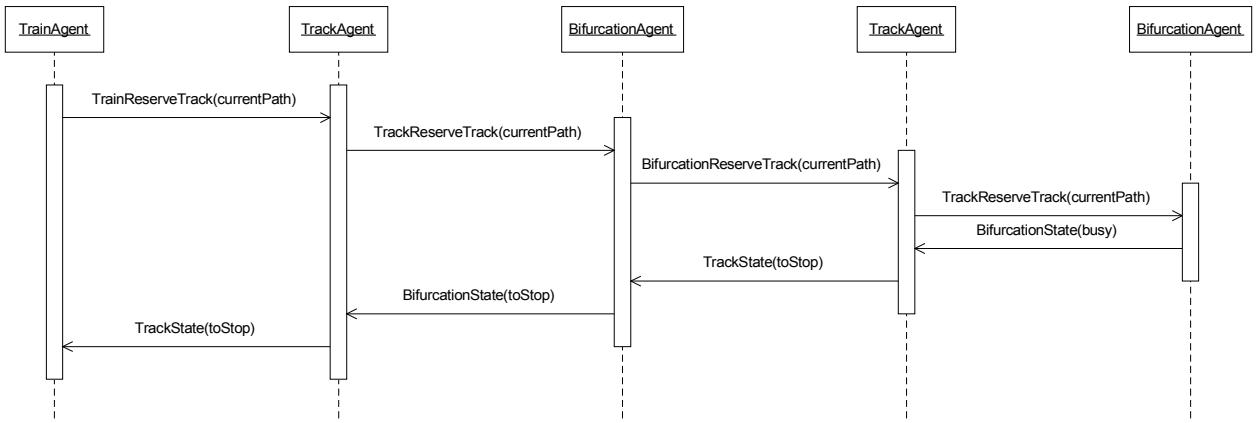

Figure 4-24: sequence diagram of the reserve track method

- The second track could not be reserved

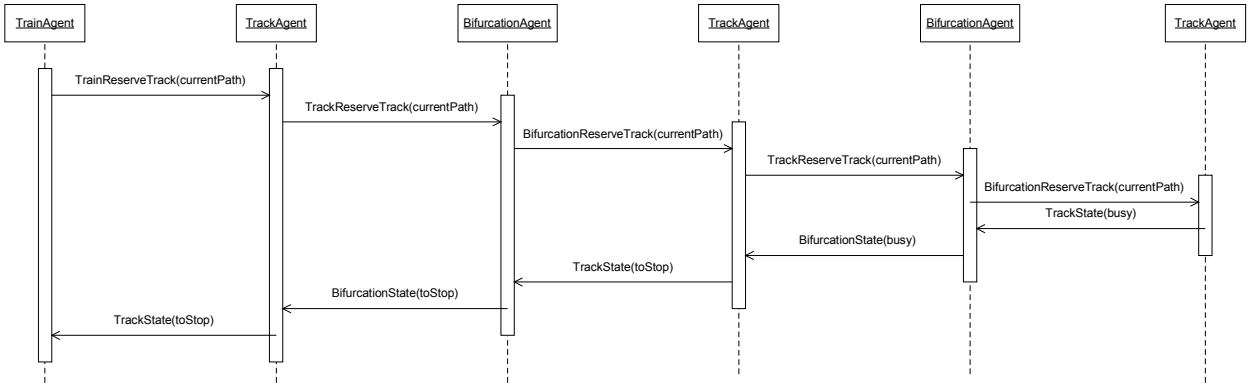

Figure 4-25: sequence diagram of the reserve track method

- The tracks and bifurcation could be reserved

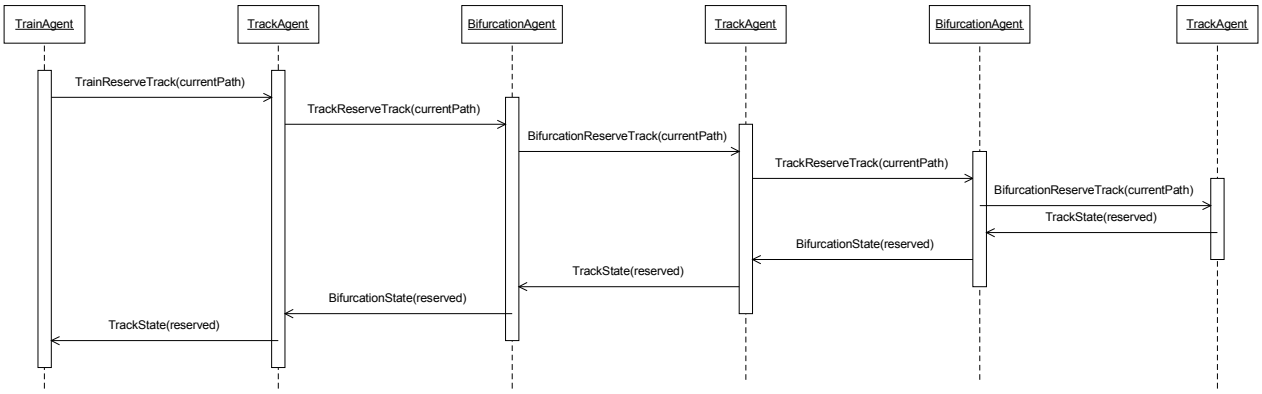

Figure 4-26: sequence diagram of the reserve track method

The **path search** method will be modeled here:

- From the train to the current track

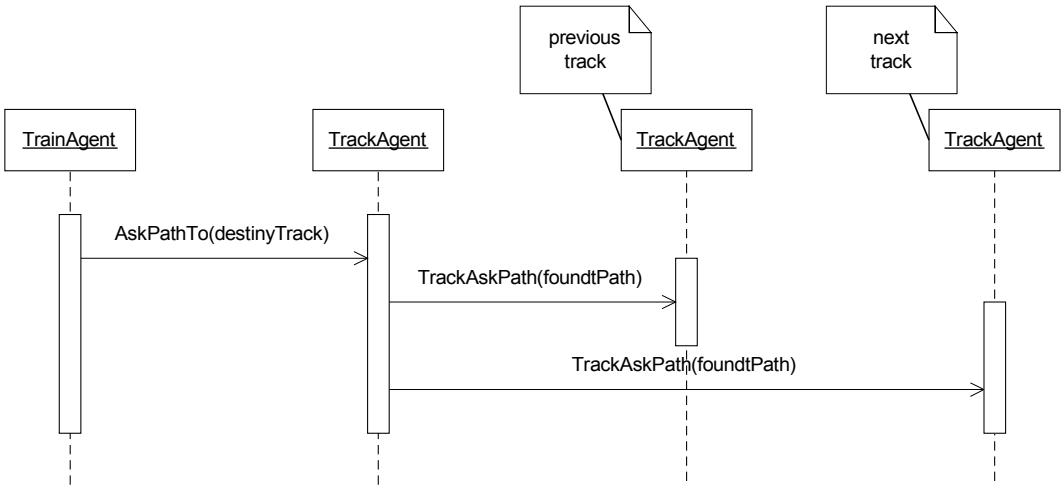

Figure 4-27: sequence diagram of the path search method

- From one track to another track

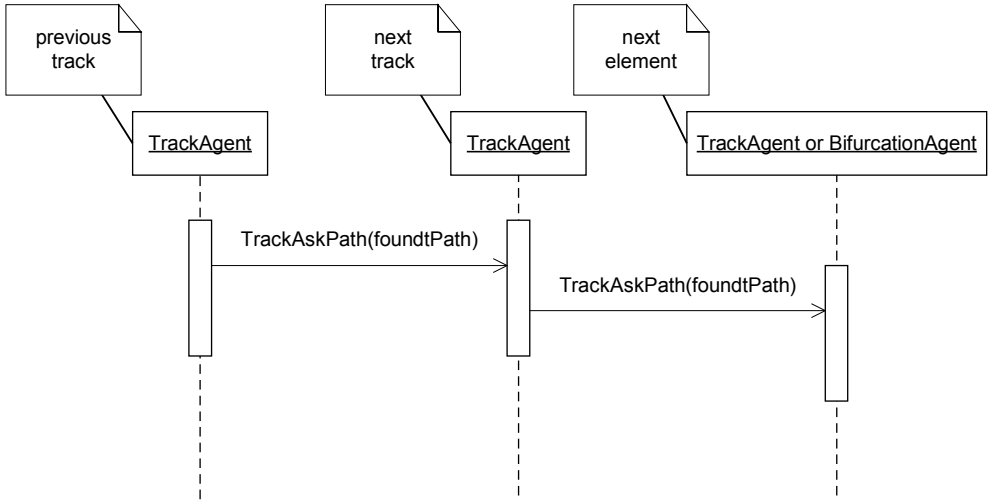

Figure 4-28: sequence diagram of the path search method

- From one track to a bifurcation

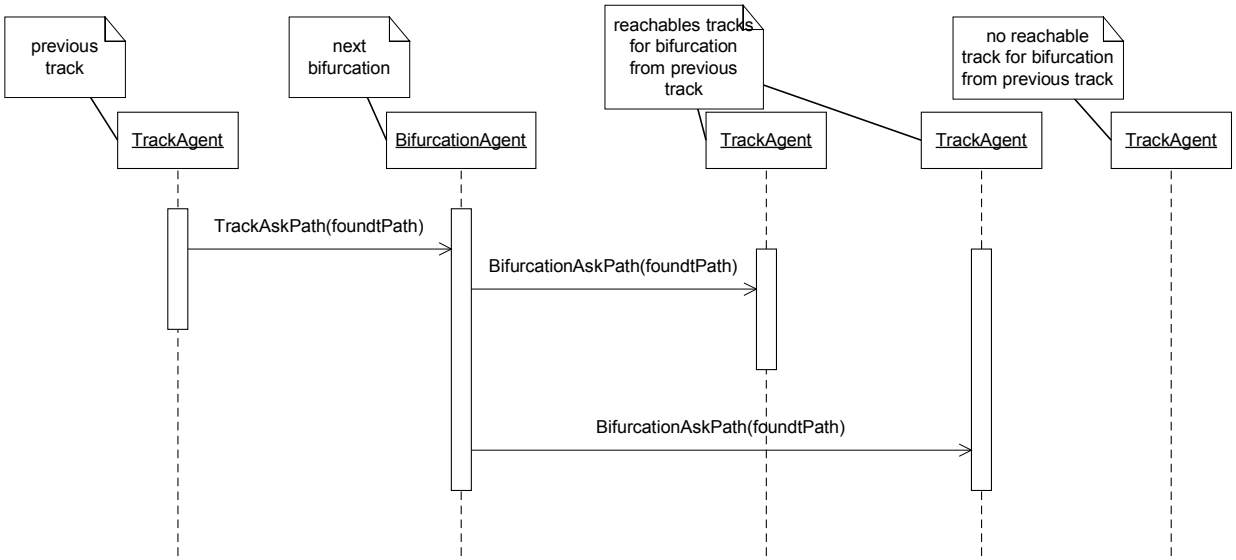

Figure 4-29: sequence diagram of the path search method

- From a bifurcation to a track

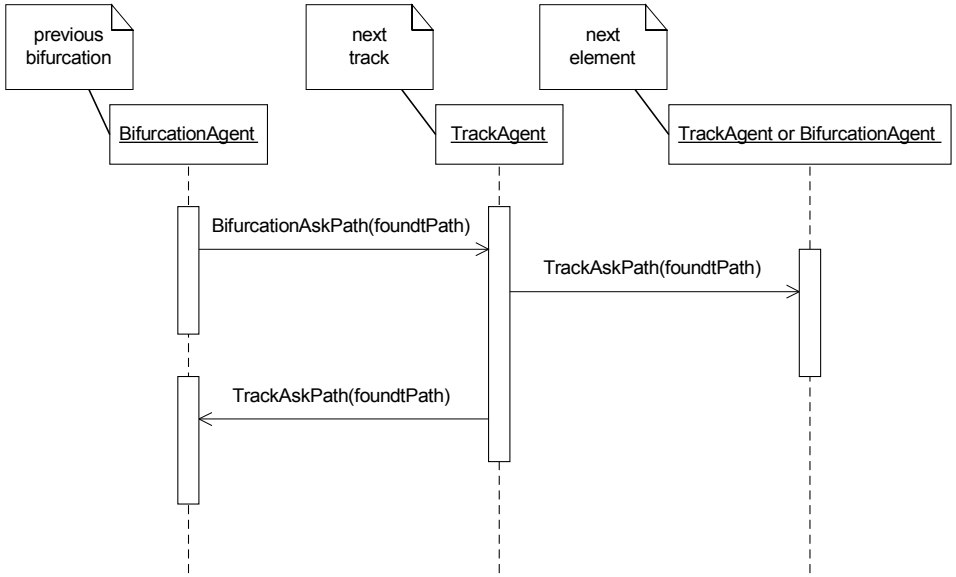

Figure 4-30: sequence diagram of the path search method

- When the destiny is arrived

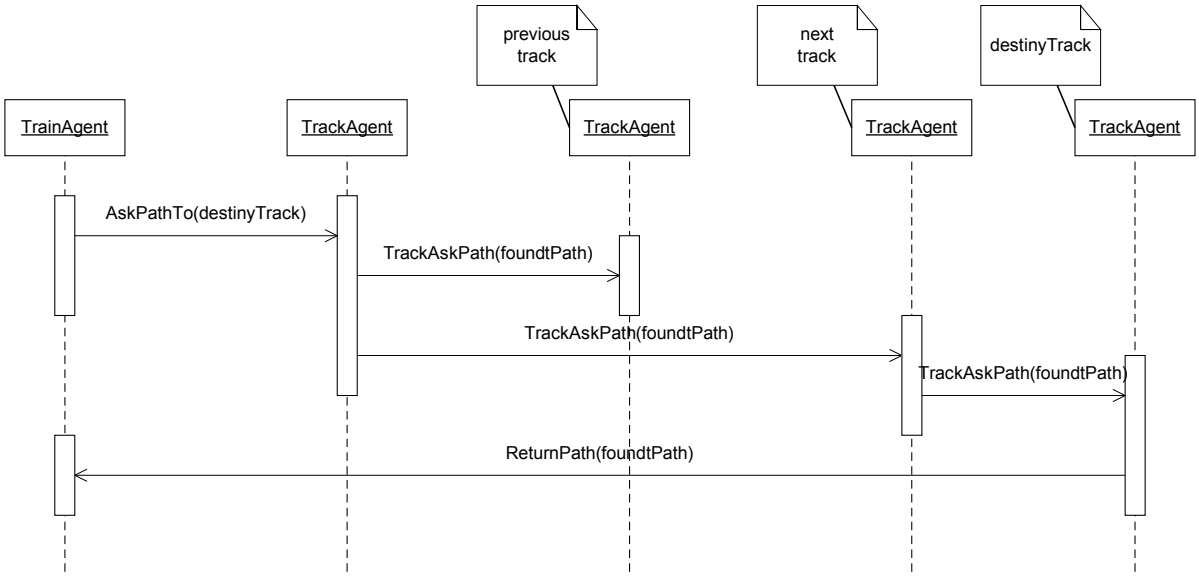

Figure 4-31: sequence diagram of the path search method

The **preference asking** and **intention spread** methods will be modeled here:

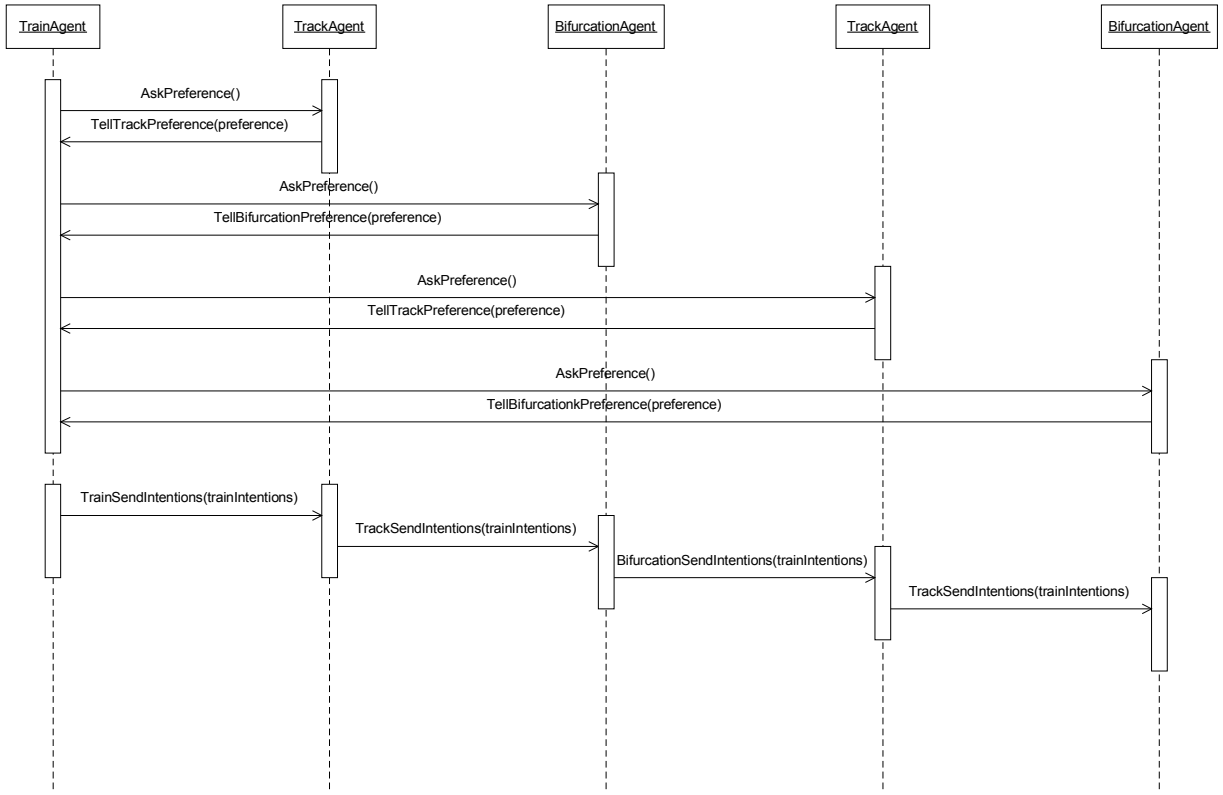

Figure 4-32: sequence diagram of the preference asking and intention spread methods

The **asking for track** of a station method will be modeled here:

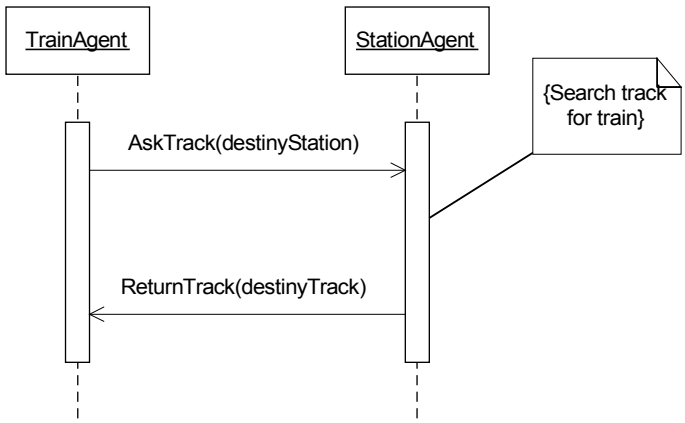

Figure 4-33: sequence diagram of the asking for track method

4.7 Summary

In this chapter the most important commands of the Intellibox protocol are explained because these commands will allow the control of the model and also impose some limitations; for example, the model is not able to tell where a train is, it is only able to know if its tracks are free or busy.

A detailed map of the model with its tracks, bifurcations and semaphores is included. This provides more insights of the problem; for example that tracks should be bidirectional, but some of them will have a preferred direction in the routes considered.

With all this information the design process of Gaia is developed. At first a decision of the roles is done taking in consideration the requirements of the project. Then the methods that will be used to accomplish theses requirements are explained in order to identify the protocols that will be used in the Interaction Model of Gaia. With the description of the protocols and the explanation of the methods, the Roles Model and the Interaction Model are created. The more difficult tasks of this step are the identification of roles and the creation of methods to accomplish the requirements; with this information, the models can be developed in a straightforward way.

The next step is the design phase of Gaia; it is also a straightforward task from the previous models. So it can be concluded that the Gaia methodology is easily applicable when the roles and the methods of the system are defined; but even in this case, the Role and Interaction Models will require some iterations to be completed.

With all the analysis and design work done, the modeling of the system with the AgentUML language is faced. This allows describing in a more detailed way the main functionalities of the system referencing the protocols described in the Interaction Model. This will provide a description of how the system will work and how the interactions of the agents will be.

5 Prototype

5.1 System Architecture

The system is composed of three different parts: the agent package, the agent manager and the railway model implementation.

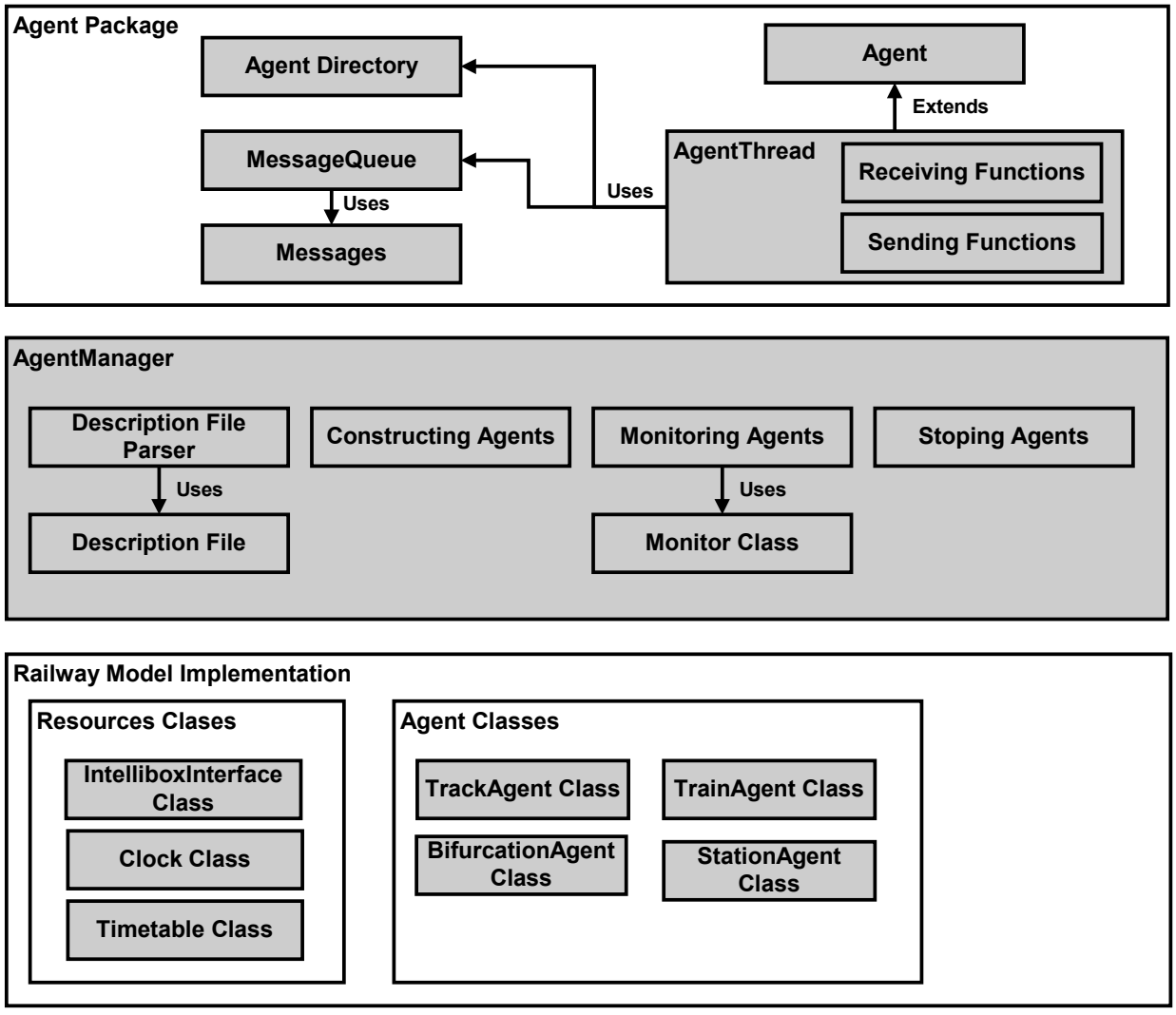

Figure 5-1: System Architecture

These different parts are explained in the following sections.

5.1.1 Agent Package

The agent package will be the agent infrastructure of the system. This will allow the creation of agents, and the communication between them. This package provides every agent with an agent directory utility that will allow sending messages to individual agents or to a group of agents. The package also provides classes for creation messages and a MessageQueue class that provides a message buffer for each agent. At last diverse functions for sending and receiving messages are implemented.

5.1.2 Agent Manager

The agent manager will use the agent infrastructure created by the agent package and will create the system infrastructure creating, monitoring and stopping the agents. The Agent Manager will be able to load a file with the specification of the railway model and then create the appropriate agents and resources that will be needed for that specification. When all the agents are created, the agent manager will start the agents and will monitor them using a Monitor class; this will allow receiving and sending messages to the system using the console. At the end, all the agents will be stopped and the program will finish.

5.1.3 Railway Model Implementation

The resources and agent implementation is done using the results from the Gaia analysis and design process. This implementation consists of the classes that the agent manager will have to create and use after loading the configuration file. These classes will perform the system functionality.

5.2 Description of the System Components

5.2.1 Agent Package

The Agent Package is composed of the following elements:

5.2.1.1 A Message Class

The Message class will be used to create the messages that the agents will send to communicate to other agents. The message consists of three parts: the sender name, the receiver name and the content. It is also possible to add a fourth part which will be the “redirectTo” which will be useful when the message is sent to an agent only for redirect purposes. The content of the message will be a string. In this content, tokens are separated by a space or by the symbol “:”. The first token will be the command and the following ones will be the arguments. For example,

in the content “ReturnPath t1:t2:t3” the command is “ReturnPath” and it will have three arguments: “t1”, “t2” and “t3”. The class has also functions to get the command and arguments of the content part of the message.

The collaboration diagram of the message is shown here (the gray boxes are standard Java API classes or types):

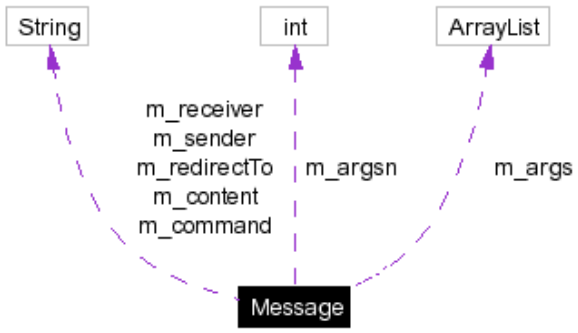

Figure 5-2: Collaboration diagram of Message class

This class will have two constructors:

- **Message**(String sender, String receiver, String content)

  This constructor will create a message with information about the sender, the receiver and the content. This constructor will be used by the AgentThread class when it will try to send a message and will fill automatically the sender argument.

- **Message**(String sender, String receiver, String redirectTo, String content)

  This constructor will be used when a message is sent to an Agent in order to that this Agent will resend this message to another Agent. It is done in the Monitor class.

And the following public member functions:

- String **sender**()

  This function will return the name of the sender of the message.

- String **receiver**()

  This function will return the name of the receiver of the message. This receiver name may be different from the name of the Agent that receives the message because if a message is sent to a group of agents, then the receiver’s name will be the name of the group and not the name of the individual agent.

- String **content**()

  This function will give the full content of the message.

- String **redirectTo**()

  This function will give the name of the agent to which this message must be resent.

- String **contentCommand**()

  This command will give the command of the content of the message.

- int **contentNoArgs**()

  This command will give the number of arguments of the content. If there are no arguments it will return 0.

- String **contentArgAt**(int i)

  It will return the argument at a specific position. The first argument will be at the 0 position. If an argument that does not exist is asked, the function will return an empty String.

5.2.1.2 A MessageQueue Class

The MessageQueue class will be used to store messages in the agents until a proper time for their process is reached. It allows creating queues with and without losses. It also allows introducing messages, getting the next message, the next message from a sender or checking if there is a message from a sender. This class has the following collaboration diagram:

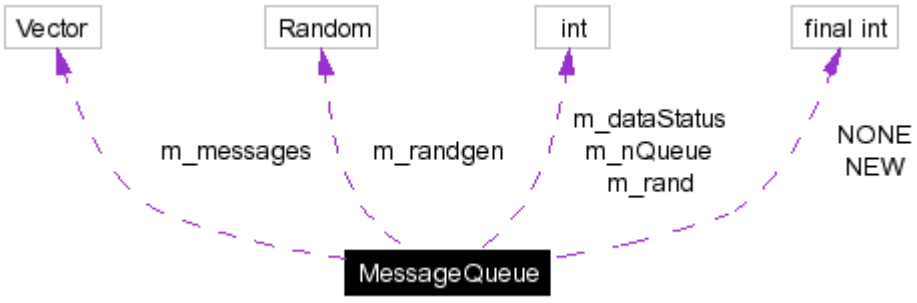

Figure 5-3: Collaboration diagram of MessageQueue class

This class will have two constructors:

- **MessageQueue**(int loss)

  This constructor will create a MessageQueue with a loss probability. The loss argument will be an integer from 0 to 100 which will represent the loss probability; 0 will be no loses and 100 that every message will be lost.

- **MessageQueue**()

  This constructor will create a MessageQueue without losses. It is the same as MessageQueue(0); but this function is provided because it is often used.

And the following public member functions:

- int **status**()

  This function will reply with the status of the Queue

0	Empty queue
1	Queue with elements

- void **insert**(Message mess)

  This function will insert the Message of the argument in the Queue

- Message **get**()

  This function will extract the first Message from the Queue. It is a FIFO queue (First In First Out)

- Message **get**(String sender)

  This function will extract from the Queue the first Message with a specific sender.

- boolean **check**(String sender)

  This function will check if there is a Message from a specific sender in the Queue.

5.2.1.3 A Directory Class

This class will allow finding other agents or group of agents. This will be helpful for sending messages to other agents or groups of them. This class will consist in a directory of objects hierarchical structured. In this class elements can be saved with a classification in the way: “class: subclass: subsubclass: …: name” this classification is not required to be strictly

hierarchical. For example to specify a passenger train the following can be used: “train: small: passengers: train1” or “train: big: passengers: train1”.

You can use the name "all", which is at the top of the hierarchy. It is possible to add elements without classification, elements without name, or both. But they will be accessible by asking for the class “all”. This class is not optimized for adding and removing objects, but for getting all objects from a name or hierarchy.

The collaboration diagram of this class is shown here:

Figure 5-4: Collaboration diagram of Directory class

The Directory class has a constructor:

- **Directory**()

  This will construct an empty Directory

And the following public member functions:

- void **add**(Object element, String classification)

  This function will add a new element to the Directory following the provided classification.

- void **remove**(String name)

  This function will remove an element or a group of elements by its name.

- ArrayList **get**(String id)

  This function will return an ArrayList with the elements that coincide with the id provided, which can be a name or a classification.

5.2.1.4 An Agent Class

This class will store the basic information of an Agent: its name and its hierarchical type. The inheritance and collaborative diagrams are shown here.

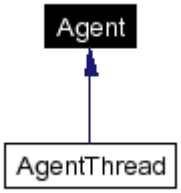

Figure 5-5: Inheritance diagram of Agent class

Figure 5-6: Collaboration diagram of Agent class

This class will have a constructor:

- **Agent**(String name, String type)

  This will construct an Agent with the provided name and a classification or type.

And the following public member functions:

- String **name**()

  This will return the name of the Agent.

- String **type**()

  This will return the type of the Agent.

5.2.1.5 An AgentThread Class

This is the base class for agent threads that allows agent communication. Every agent will implement this class and define the functions processInit(), processLoop(), processEnd() and receivedMessage(Message mess). It allows the use of createTimer(int timeout) in the processInit() function, the use of receive(), receive(int timeout), receive(String sender) and receive(String sender, int timeout) in processInit() and processLoop(). It also allows to get a message at the moment it is received in the receivedMessage(Message mess) function. The

function processEnd() will be used for finalization processes. You can send a message with send(String receiver, String content) anywhere in the agent. The inheritance and collaborative diagrams are shown here.

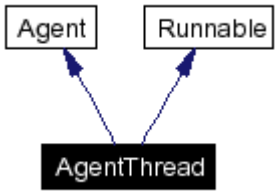

Figure 5-7: Inheritance diagram of AgentThread class

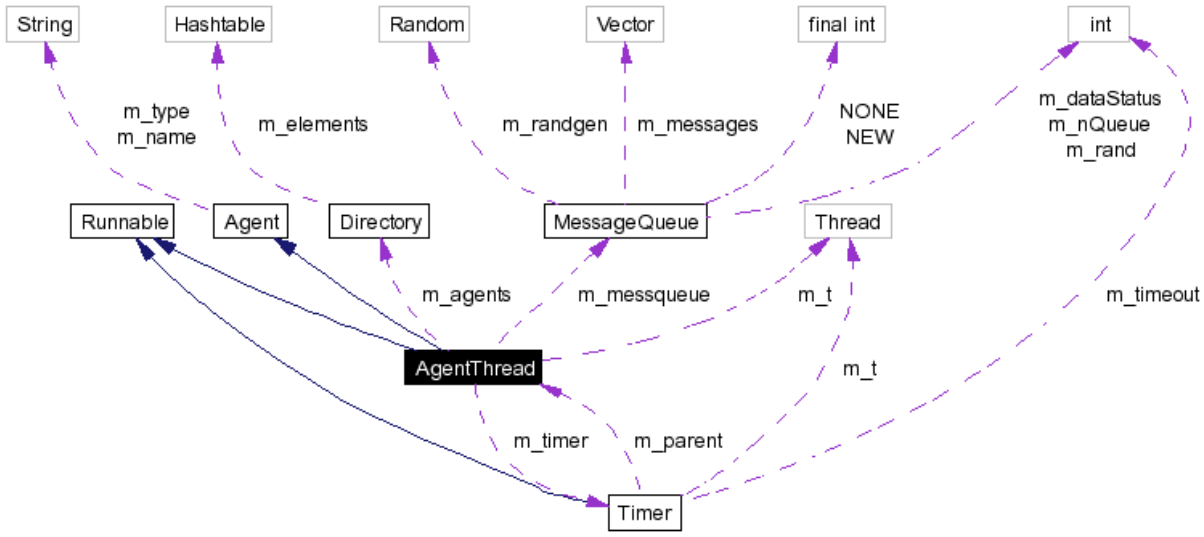

Figure 5-8: Collaboration diagram of AgentThread class

This class has a constructor:

- **AgentThread**(String name, String type)

  It will introduce the agent in the directory, that will be a static variable accessible to all the AgentThread instances and create its message queue.

And the following public member functions:

- void **start**()

  It will start the thread of the agent.

- void **run**()

  This is the main loop of the thread. It will call processInit() once at the beginning and then it will call processLoop() repeatedly until the stop function is used and then it will call processEnd().

- void **stop**()

  It will stop the agent and the timer of the agent. Also it will erase the directory entry of this agent. This is a **private** function, but it will be called automatically when a message with a "stop" command arrives to the agent.

- Message **receive**()

  This function will wait until the agent has a message in the queue. It will return the first message in the message queue.

- Message **receive**(int timeout)

  This will wait until the agent has a message in the queue or the timeout expires. It will return the first message in the message queue.

- Message **receive**(String sender)

  This will wait until the agent has a message in the queue from a specific sender. It will return the first message in the message queue from the specified sender.

- Message **receive**(String sender, int timeout)

  This will wait until the agent has a message in the queue from a sender or the timeout expires. It will return the first message in the message queue from the specified sender.

- void **send**(String receiver, String content)

  This will send a message from the calling agent to the receiver. A message will be created by filling the sender field with the correct value.

- void **createTimer**(int timeout)

  It will create a Timer that will continuously send a message to the agent every time the timeout expires. This message will have the content "Timer".

- void **receive**(Message mess)

  This allows the agent to receive messages. This function is used by other agents so that this agent receives a message and put it in the message queue. It will call the

receivedMessage(Message mess) function. It also stops the agent if it receives a "stop" message.

- void **processInit**()

  The initialization of the agent is done here. This function will be implemented in the agent and will be called when the agent has been started.

- void **processLoop**()

  The main loop of the agent is performed here. This function will be implemented in the agent and will be called repeatedly.

- void **processEnd**()

  This function fill be called at the end of the agent execution. This function will be implemented in the agent and will be called at the end of the agent execution.

- Message **receivedMessage**(Message mess)

  When the agent receives a message, this function will be immediately called. This function will be implemented in the agent and will be called each time the agent receives a message. The message that will be added to the queue will be the one returned by the function. If it returns null, no message will be added. The stop message will be processed after the return of this function so it is possible to process the “stop” message to avoid the stop of the agent.

- String **name**()

  This returns the name of the agent.

- String **type**()

  This returns the hierarchical type of the agent.

5.2.2 Agent Manager

The Agent Manager is composed of the following elements:

5.2.2.1 A Descriptor File

The description of the railway model, trains position and timetables can be stored in one or several files. A possible structure might be to save the physical railway model definition in a

file; the description of trains and their positions in another file, and the timetables in another file. But any combination will be possible in the program.

In these files blank lines can be inserted between any line of the description. Also lines with comments can be inserted if their first character is "%"

The following elements can be described in these files:

- **Tracks**

  The tracks will be described as follows:

  track trackName

  moduleAddress modulePosition

  A nextItem

  B prevItem

  AS semaphoreToNextItemAddress

  BS semaphoreToPrevItemAddress

  The AS and BS sections are optional. The A and B sections are needed, but if one of them does not proceed, it can be followed by nothing:

  A nextItem

  B

- **Bifurcation**

  The bifurcations will be described as follows.

  bifurcation bifurcationName

  %Addresses of real bifurcations separated by spaces

  1 2 3 4

%Names of the tracks that the bifurcation connects

t6 t7 t8 t9

Track1 -> Track2 : bifurcation1Address state : bifurcation1Address state

…

If from Track1 to Track2 the train uses a bifurcation but it is no necessary to put it to a specific state, the state “x” can be used.

- **Stations**

  The stations will be described as follows:

  station stationName

  %Tracks of the station separated by spaces

  t6 t7 t8 t9

  Platform1 track1Name

  …

- **Trains**

  The trains will be described as follows:

  train trainName

  trainAddres frontDirection

  currentTrainTrackName

  direction

  The “frontDirection” is the value that will be used in the Intellibox commands to send the train in forward direction; it can be 1 or 0. The “direction” is the direction of the train

over the track; it will be “A” if the train head is in the direction of the next track of its current track and it will be “B” otherwise.

- **Timetables**

  The timetable will be defined in the following way:

  timetable timetableName

  train destinyStation hh:mm

  …

  It can be defined in every order with the exception that for a same train the definitions must appear in correct time order. For example the following is correct:

  timetable table1

  train1 destiny1 11:00

  train2 destiny2 10:00

  train1 destiny3 11:30

- **Clock**

  The clock will be defined in the following way:

  clock clockName

  %Start time of the clock

  hh:mm:ss

  The velocity of the clock is faster than real clocks. Every three real seconds it will advance one minute.

5.2.2.2 An AgentManager Class

The AgentManager will have the following tasks:

- Read the descriptor file.

  The class will open all the files specified in the arguments of the program call and will parse them to extract the correct information. If an item is incorrectly defined, it will be avoided and the next items will be processed.

- Create the needed class from the information of the descriptor file.

  The needed agents for the railway model described will be created and started.

- Create a IntelliboxInterface class and a Monitor class.

  A class that will control the Intellibox will be created and started.

- Read from standard input and send the messages to the Monitor class.

  The program will read from standard input. It will assume the first line will be the receiver of a message and the second one the content of the message. The program will create a message and will send it to the Monitor class.

- When a stop command is introduced, stop all the agents.

  When a “stop” command is introduced, the program will send a “stop” message to all the agents so that they will be stopped.

5.2.2.3 A Monitor Class

The Monitor class will allow the user to send and receive messages from the agents at runtime. The AgentManager class will receive commands from the standard input and then they will be sent to the Monitor with the “resendTo” argument with the name of the Agent the message is intended to be sent. The Monitor class will resend the message to the correct Agent and if the Agent responds to this message to the Monitor class, it will show the reply on the standard output.

5.2.3 Railway Model Implementation

The Railway Model Implementation is composed of the following elements:

5.2.3.1 A Clock Class

This class will have the following constructor:

Clock(String name, int hour, int min, int sec)

This will create a clock initialized to a certain time. The clock will accept messages with the "askTime" commands and will reply with the time in the format "hh:mm:ss"

5.2.3.2 A IntelliboxInterface Class

This class will be used as an interface to the Intellibox controller. So this will be the only class that will send messages to the Intelibox. It will not send the messages directly, but it will create a Socket communication at the port 2003 and will send the messages to a interface program that will redirect the messages to the serial port and then to the Intellibox controller.

This class will have two main tasks:

- When a message arrives to this class it will send it to the interface program.
- At certain intervals it will check the status of the tracks and if they have been modified, a message will be sent to the corresponding track informing about its new state.

5.2.3.3 A Timetable Class

This class will provide the Trains with the necessary information about the introduced timetable.

5.2.3.4 A TrainAgent Class

This class will create a TrainAgent and will process the protocols defined in the Conception document. This agent will be in charge of try to follow the assigned timetable in a safe way.

5.2.3.5 A TrackAgent Class

This class will create a TrackAgent and will process the protocols defined in the Conception document. In addition it will allow the creation of logical tracks. That means, tracks with the same address in the railway model but that will react as different tracks in the program. The only limitation is that all the logical tracks will be reserved at once as if they were the same track. This agents with the BifurcationAgents will find paths for trains and will allow them to move in a safe way.

5.2.3.6 A BifurcationAgent Class

This class will create a BifurcationAgent and will process the protocols defined in the Conception document. This agent also will put the bifurcations to the correct state to allow trains to move to the correct track.

5.2.3.7 A StationAgent Class

This class will create a StationAgent and will process the protocols defined in the Conception document. This agent will be in charge of assigning tracks for the arrival of trains to the station.

5.2.4 Improved characteristics

The program has more characteristics that the ones described in the Conception document:

- The program will detect when there is a obstacle in a track; and if it is not a train, the program will consider this track to be broken. So this track will not be taken in account to find paths.
- The program will detect when a train is not moving and it will consider that the train is breakdown. This sometimes happens because the train lost contact in the model. The program will show a message indicating which train does not respond. When a train is considered to be breakdown, then the current track of this train will be considered as broken and it will not be taken in account to find paths. When the train begins to move again the track will recover its normal value.
- Sometimes a wagon is lost from a train. The program will detect it and will consider the track when the wagon was lost as broken. A message will be shown to indicate which train has lost a wagon and on which track the wagon is. When the wagon is removed, the will track recover its normal value.

All this improvements have not been considered in the analysis and design phases, because of the structure of the program, they were very easy to implement using the data of the current agents.

5.3 Installation and User Manual

The following steps are required to run the program:

- First, the computer must be connected to the Intellibox controller of the railway model using a serial cable.
- Then the Interface program must be started its name is “IBSerP50X2.exe” and can be found in the directory “InterfaceProgram” of the prototype.
- The CLASSPATH variable must be set in a way that allows loading the “agent” package.

  At the end, the java program must be executed using:

java AgentManager descriptionFile1 descriptionFile2 …

Where the descriptions files are the ones used for describe the model. The prototype is provided with the following description files:

- map.def: describes the physical model at the IAS
- trains.def: defines the trains of the model and its positions. It must be modify to describe where the trains are at the start position.
- timetable.def: defines a timetable to move the trains.

- Then commands can be introduced in the program; the first line will be the receiver of the message and the second line the message. To allow trains to begin to follow their timetable the following can be sent:

  Train <Enter>

  Go <Enter>

- Now the program is running and more commands can be inserted using the console. The more useful ones are:
  - To send direct command to the intellibox it possible to type:

    ii <Enter>

    intelliboxCommand <Enter> (do not use the initial "x" of the command)
  - To send a train to a destiny, type:

    trainName <Enter>

    trainGoTo destiny <Enter>
  - To set a bifurcation to a correct state, type:

    bifurcationName <Enter>

    set originTrack destinyTrack <Enter>
  - To stop a certain agent, type:

    agentName <Enter>

    stop <Enter>

- To stop the program, type:

  stop <Enter>

  <Enter>

The used Java version for the development of this program was 1.4.

6 Conclusion and Outlook

6.1 Summary

In this project at first, I have learnt about the agent concept and the most important agent oriented methodologies. This has improved my background in software development and let me know about certain aspects that are very interesting to me.

With these concepts I have begun the implementation of the agent infrastructure of the prototype, which at first has lots of limitations, but in the end all the flexibility characteristics of an agent oriented system were implemented.

Then I accomplish the analysis and design phases of the project with the chosen methodology (Gaia) but the phase of analysing the different kinds of communication between agents was very difficult and required a lot of creativity.

At the end all this design was implemented in the prototype with good results.

6.2 Experiences

For the implementation of agent oriented systems it is needed an agent infrastructure that I have implemented in this project. It requires a large initial effort but it is supposed that it can be reused for latter projects. So I consider that agent oriented development at first will require some additional effort, but in latter projects previous work can be reused and improved, allowing easier start points in the future.

The infrastructure of an agent oriented system represents indeed an overhead if compared with a system with similar functionality developed with another approach. Nevertheless, it has revealed that the agent infrastructure has two advantages:

This infrastructure will allow a more flexible system, which will be very difficult to achieve using traditional methodologies.

Also, the developing of this infrastructure will be done once and then can be much optimized, so it can be even less overloading that using traditional methodologies.

Also I have experienced the great power of agents when I have implemented the prototype. That is because even when the prototype was not completely implemented, the trains could resolve

situations not considered in their implementation. They took more time than when they were fully implemented, but at the end they resolved their conflicts.

The agent oriented implementation has also shown to be very easy to be modified, because when I needed to change some of the important mechanism of their functionality, it was very easy to know where to do those modifications and there were no interdependencies with other functions or mechanisms; for example, it will be very easy to modify the algorithm to find paths and it will not require the modification of any other functions in the implementation.

I have also learn a lot from the IAS working model and I think it is very useful for an organized work and to provide valuable documentation for future works.

6.3 Problems

Some problems arose at the beginning because the definition of the agent concept. That is because of the variety of agent definitions, but at the end we found a proper definition that can be useful in both theory and practical purposes.

Appendix A Index of Figures

Appendix B Index of Tables

Appendix C Abbreviations

AAII	Australian Artificial Intelligence Institute
AI	Artificial Intelligence
AOR	Agent Object Relationship
AOSE	Agent Oriented Software Engineering
IAS	Institute of Industrial Automation and Software Engineering
MASSIVE	MultiAgent SystemS Iterative View Engineering
MESSAGE	Methodology for Engineering Systems of Software Agents
OMT	Object Modelling Technique
OO	Object Oriented
OPEN	Object-oriented Process, Environment and Notation
PC	Personal Computer
RUP	Rational Unified Process
SODA	Societies in Open and Distributed Agent spaces
UML	Unified Modeling Language

A Appendix D Terminology

Acquaintance Model	Description of the communication channels among agents
Actuators	Mechanisms that allow agents to act in an environment.
Agent	An abstraction used for software development
Agent Model	Description of the roles that an agent can take
Interaction Model	Description of the protocols of a system
Protocol	The way agent can communicate between them
Resource	Means that provide information or allow to be modified by agents
Roles	The tasks that an agent can take
Roles Model	Description of the roles of a system
Sensors	Mechanisms that provide information to agents
Sequence Diagram	UML diagram showing how the protocols are send and processed
Services Model	Description of the main services of a system

Appendix D Literature

[1] **R. C. Lee, W. M. Tepfenhart.** *Practical Object-Oriented development with UML and Java.* Prentice Hall.

[2] **T. Kuhn.** *The Structure of Scientific Revolution.*

[3] **G. Weiss.** *Multiagent Systems*. MIT Press. 1999

[4] **N. R. Jennings, K. Sycata, M. Woodridge.** *A roadmap of agent research and development*. Int. Journal of Autonomous Agents and Multi-agent Systems. 1 (1), pp7-38

[5] **S. Russell, P. Norvig.** *Artificial Intelligence: A Modern Approach*. Prentice Hall. 2003

[6] **B. Hayes-Roth.** *An architecture for adaptative intelligent systems.* Artificial Intelligence: Special Issue on Agents and Interactivity 72. 1995. pp329-365.

[7] **M. Wooldridge, N. Jennings**. *Agent theories, architectures and languages: A survey*. Intelligent Agents, Agent Theories, Architectures and Languages (ATAL 94), vol 890 Springer-Verlag Lecture Notes in Artificial Intelligence. 1994. pp1-32

[8] **B. Henderson-Sellers, I. Gorton.** *Agent-based Software Development Methodologies*. http://www.open.org.au/Conferences/oopsla2002/Whitepaper.pdf. 2002.

[9] **K. Hoa Dam**. *Evaluating and Comparing Agent-Oriented Software Engineering Methodologies.* http://yallara.cs.rmit.edu.au/~kdam/Hoathesis.pdf.

[10] **J. Odell**. *Modeling-Notation Source: AOR*. http://www.auml.org/auml/documents/AOR.pdf.

[11] http://AOR.rezearch.info.

[12] **M. Cossentino, C. Potts**. *PASSI: a Process for Specifying and Implementating Multi-Agent Systems Using UML*.

[13] *Wikipedia* http://www.wikipedia.org/wiki/Object_modeling_language.

[14] *OO Software Architecture Fusion Methodology.* http://www.cs.ualberta.ca/~hoover/Courses/cmput401-200004/SoftwareArch/attach/fusion.htm.

[15] **M. Wooldridge, N. R. Jennings, D. Kinny.** *The Gaia Methodology for Agent-Oriented Analysis and Design.*

[16] *Contrast and comparison of five mayor Agent Oriented Software Engineering (AOESE) methodologies.* http://students.jmc.ksu.edu/grad/madhukar/www/professional/aosepaper.pdf.

[17] **A. Omicini.** *SODA: Societies and Infrastructures in the Analysis and Design of Agent-based Systems.*

[18] **Xiaobing Qiu.** *Object-Oriented Software Development Using Object Modeling Technique (OMT).*

[19] **F. Vermaut, J. C. Trigaux, P. Y. Schobbens.** *Extending the AAII methodology for programming agents in 3APL.*

[20] **S. A. Deloach, M. F. Wood, C. H. Sparkman.** *Multiagent System Engineering.*

[21] **B. Henderson-Sellers, H. Younessi.** *Introducing Object Technology through the Use of the OPEN Methodology.* http://hsb.baylor.edu/ramsower/ais.ac.97/papers/hendsel.htm.

[22] **R. Evans, P. Kearney, J. Stark, G. Caire, F. J. Garijo, J. J. Gomez Sanz, J. Pavon, F. Leal, P. Chainho, P. Massonet.** *MASSE: Methodology for Engineering Systems of Software Agents. Methodology for Agent-Oriented Software Engineering (final)*. 2001

[23] **C. Bernom, M.-P. Gleizes, S. Peyruqueou, G. Picard**. *ADELFE, a Methodology for Adaptative Multi-Agent Systems Engineering.*

[24] **L. Padgham, M. Winikoff.** *Prometeus: A Methodology for Developing Intelligent Agents.*

[25] **L. Padgham, M. Winikoff.** *Prometeus: A Pragmatic Methodology for Engineering Intelligent Agents.*

[26] **A. Drogoul, J.-D. Zucker**. *Methodological Issues for Designing Multi-Agent Systems with Machine Learning Techniques: Capitalizing Experiences from the RoboCup Challenge.*

[27] **F. Giunchiglia, J. Mylopoulos, A. Perini.** *The Tropos Software Development Methodology: Processes, Models and Diagrams.*

[28] **J. Lind.** *MASSIVE: Software Engineering for Multiagent Systems.*

[29] **A. Omicini.** *From Objects to Agent Societies: Abstractions and Methodologies for the Engineering of Open Distributed Systems.*

[30] http://www.andrew.cmu.edu/user/conzalez/Teaching/ISW2/OMTintro.html

[31] *Distributed Artificial Intelligence and Intelligent Agents.* http://www.idi.ntnu.no/~agent/

[32] http://www-poleia.lip6.fr/~drogoul/projects/cassiopeia/

[33] **P. Maes.** *Artificial Life Meets Entertainment: Life like Autonomous Agents.* Communications of the ACM, 38, 11, 108-114. 1995

[34] http://www.rjftrains.com/intellibox/intellibox.htm