Dipartimento di Informatica e
Scienze dell'Informazione

CaseLP, a Rapid Prototyping Environment for Agent-Based Software

by

Floriano Zini

Theses Series **DISI-TH-2001-03**

DISI, Università di Genova
v. Dodecaneso 35, 16146 Genova, Italy http://www.disi.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica e Scienze dell'Informazione

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

CaseLP, a Rapid Prototyping Environment for Agent-Based Software

by

Floriano Zini

June, 2001

Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova

DISI, Univ. di Genova
via Dodecaneso 35
I-16146 Genova, Italy
http://www.disi.unige.it/

Ph.D. Thesis in Computer Science

Submitted by

Floriano Zini
DISI, Università degli Studi di Genova, Italy
zini@disi.unige.it

Date of submission: December 2000

Title:

CaseLP, a Rapid Prototyping Environment for Agent-Based Software

Advisor:

Prof. Maurizio Martelli
DISI, Università degli Studi di Genova, Italy
martelli@disi.unige.it

Ext. Reviewers:

Prof. Alberto Martelli
Dipartimento di Informatica
Università degli Studi di Torino, Italy
mrt@di.unito.it

Prof. Leon Sterling
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
leon@cs.mu.oz.au

Abstract

Intelligent agents and multi-agent systems are increasingly recognized as an innovative approach for analyzing, designing and implementing complex, heterogeneous and distributed software applications. The agent-based view offers a powerful and high level conceptualization that software engineers can exploit to considerably improve the way in which software is realized. Agent-based software engineering is a recent and very interesting research area. Due to its novelty, there is still no evidence of well-established practices for the development of agent-based applications and thus experimentation in this direction is very important.

This dissertation presents CaseLP (*Complex Applications Specification Environment based on Logic Programming*), an experimental environment based on logic programming for rapid prototyping of agent-based software applications. CaseLP provides developers with a prototyping method and a set of tools and languages which allow for the realization of an executable prototype of a software application as a society of interacting agents.

A prototype is built along three dimensions: task organization, knowledge organization and interaction organization. Task organization concerns the specification of tasks performed by the prototype, the distribution of tasks among agents, the services that agents provide and request to accomplish their tasks. CaseLP provides tools for task organization: an architectural description language is adopted to describe the prototype in terms of agents classes, instances, their provided and required services, and their communication links. Knowledge organization concerns the description of knowledge that agents must manipulate and use in order to behave properly in order to execute their services. Ontologies are used in CaseLP to explicitly structure the knowledge within the prototype's application domain, as well as to unambiguously describe characteristics and properties of agents. A rule-based language is used to define procedural knowledge of reactive and/or proactive agents. Interaction organization concerns the way agents interact with each other. A tool for conversation description is used in CaseLP to define the sequences of messages that are exchanged to provide any service, and the hierarchical relationships among sequences. Prototypes are implemented in an extended Prolog-like language. Visualization and simulation tools are included in CaseLP to visualize and trace a prototype execution and to collect statistics on it.

The dissertation also investigates the use of high level specification languages to describe sophisticated agent architectures. A linear logic language is used to specify a BDI architecture. An imperative language for the specification of heterogeneous agent architectures is

presented and discussed. These two languages have still to be fully integrated into CaseLP but represent good candidates to be used in the CaseLP method when the specification of specialized agent architectures is needed.

The research on CaseLP has been undertaken with two main objectives. Our first goal was showing that logic programming facilitates the realization of an environment for the development of agent-based prototypes. The second goal was to demonstrate that such an environment can be effectively used for the rapid prototyping of complex real-world software applications. We argue that CaseLP is a good example of how the first goal can be satisfied; the prototyping, in CaseLP, of some real-world case-studies is the demonstration of the applicability of our environment. In particular, the dissertation presents in details KICKER, a prototype for a decision support system to be used by European traffic operators in the management of freight trains movements between inter-modal centres and yards. KICKER is a project in co-operation between the Information Systems Division of Italian Railways (FS S.p.A.) and the Computer Science Department at the University of Genova, Italy.

To my family: Annalisa, Corrado, and Mirella

Acknowledgements

First, I want to thank my advisor, Maurizio Martelli, for his support and guidance during these years.

A special thanks to Viviana Mascardi. Collaborating with her has been a real pleasure and her help has been fundamental for the realization of this thesis.

I want also to thank Giorgio Delzanno and Marco Bozzano, who contributed to part of this research work.

I had the privilege of working with Leon Sterling who invited me to the Department of Computer Science and Software Engineering at the University of Melbourne, Australia. I want to thank him for his help and kindness during my stay in Melbourne, and for teaching me something new about how to do research.

Thanks to Philip Dart, Ed Kazmierckaz, and V.S. Subrahmanian for the interesting discussions that influenced the contents of this thesis.

Thanks to all the students at DISI that contributed to this research work, in particular Marco De Pedrini and Simone Marini.

Thanks to the external reviewers, Alberto Martelli and Leon Sterling, who carefully read the thesis and gave me many helpful suggestions.

I want to thank people at ITC-irst, who gave me all the support (and time) that I needed in the last few months to conclude this work. In particular, thanks to Paolo Busetta, Luciano Serafini, and Paolo Traverso.

Many thanks to Mark Carman, for his fundamental help in improving the quality of expression of the thesis.

A friendly and warm working environment is fundamental and there are many people at DISI that I would like to thank for this: Eva, Giorgio, Davide, Viviana, Ruben, Walter, and Isa are some of them. In particular, a very special thanks to Giovanna and Gianna, for their constant helpfulness, friendship (and patience!) during these years.

Finally, I want to thank my parents, Corrado and Mirella, and my sister, Annalisa. I always felt them close to me, even if I was physically distant. *Grazie*!

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The need for adequate technologies for use in the realization of increasingly complex software applications dramatically increased in the middle of 70s, when the production of software began to be transformed into an industrial activity and its cost overtook that of hardware. In those years we saw the emergence of the *software crisis* that was characterized by three main problems:

- produced software could not fully exploit the ever increasing power of hardware;
- programs could not satisfy the demands of the market;
- poor design hindered the (reasonably cheap) maintenance of software.

As a consequence, *software engineering (SE)* [71] came into fashion as an answer to these problems. An early but still valid definition of SE was proposed by Fritz Bauer [65]:

> *software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*

One major point to be taken into consideration in software production is *scalability*. For example, a handy person can easily build a backyard shed just exploiting her/his personal capabilities, but these skills will not scale for the creation of a 40 floor office building. This means that SE is principally a discipline applied by teams to produce high quality, large-scale, cost-effective software that satisfies the user's needs, and can be maintained and extended over time.

Ideally, the activities included in a SE process should be driven by a *method* and each step in the method should be performed using a set of *tools*, which enable the software engineer to gather the client's initial requirements and to transform them into a design, a language form, and machine executable code. In other words, a method provides the guidelines for the software building process, along with tools which help the software engineer in the application of the method, hopefully in an automatic or semi-automatic way.

In order to produce correctly functioning software, *validation* and *verification* are fundamental in any software engineering method. Validation is the process in which one checks whether a software product conforms with its requirements. Verification is the process of checking whether the output of a step in a software development method agrees with its input. Verification and validation of software is in general a very complex matter, especially using formal approaches. *Formal methods* [5] may be used to specify and model the behavior of a computer software system and to mathematically verify that the system design and implementation satisfy system functional and/or safety properties. Even though these specifications, models, and verifications may be done using a variety of techniques and with various degrees of rigor, formal methods are not yet widely used to develop "real" software applications, mainly because they are difficult to use, and also require too much of a mathematical background. Furthermore, most formal approaches do not facilitate the realization of applications based on pre-existent software components (*legacy software*), because they do not allow for modelling systems in which legacy components coexist with components which are only specified. For all these reasons, a software engineering approach that permits specification and partial development of new software products based on existing ones, profitably mixing experimentation with rigor, may be the right choice to realize large scale software products.

1.1 Prototyping

Among the different paradigms for SE, we focus on *Prototyping*. A *software prototype* is generally a simplified and/or not very efficient version of an end product, that is developed to validate, test and possibly modify the initial requirements of the software application under development. Figure 1.1 illustrates the various activities that constitute prototyping. During the *requirements gathering* stage functional and non functional features of the software product to be realized are collected. On this basis a *quick design* and an *implementation* are performed. Following this, the prototype is *evaluated* by means of testing and interaction with the final user or customer and, if evaluation results are satisfactory, the end product is *engineered*. Otherwise, the requirements are *refined* and another quick design is realized. The pattern "refinement, quick design, implementation, evaluation" is repeated until there is (hopefully) convergence on a product which is ready to be engineered.

Compared to other SE paradigms, prototyping gives three main advantages in software

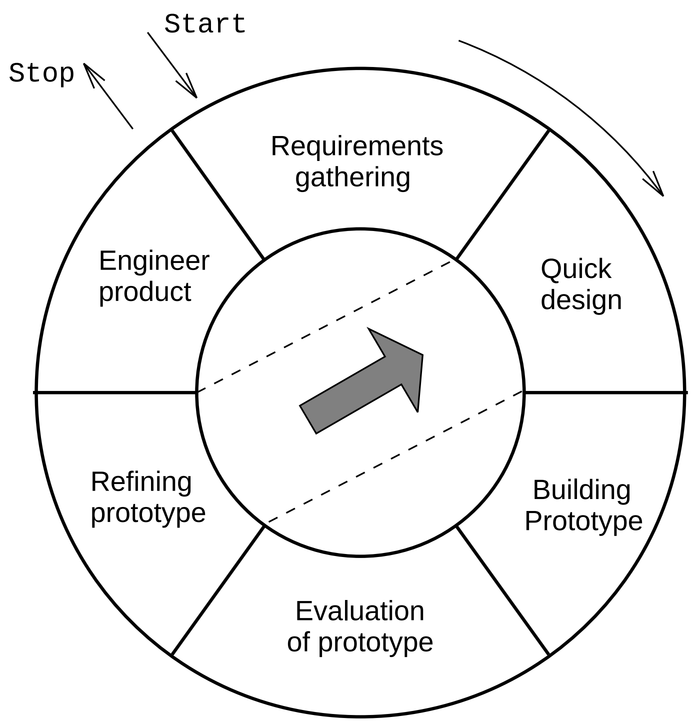

Figure 1.1: Prototyping.

realization: *rapidity*, *low cost*, and *iterative development*. Rapidity means the early availability of a product to be evaluated by the customer. This allows early detection of possible inaccuracies in software requirements and their timely modification. Furthermore, the cost of a software prototype is generally much lower than the cost of an end product and this dramatically reduces the money risks concerning the development process. Finally, prototyping is an iterative process and this gives the software engineer the flexibility to revise requirements or design choices several times and therefore to avoid premature decisions on critical features of the software product.

Another advantage is that a prototype does not usually need to be very effective and therefore it can be realized using more formal, even though less efficient, methods and tools. This generally allows for the realization of a software prototype following quite a formal approach, even if it is not feasible to adopt formal techniques for the realization of the final version of the product.

1.2 A promising combination for the prototyping of innovative software

Despite thirty years of research and experience and many successful results, software systems are still difficult to engineer so as to guarantee correctness and reliability. This is particularly true for *distributed* software systems where a set of entities have to cooperate and coordinate in order to exchange information. Many distributed systems must make use of existing software modules and consequently must *integrate* information from a potentially large number of different sources. Thus, integration and reuse of different kinds of information and software tools is an urgent necessity that new software products must address.

Obviously, the realization of reliable software applications that reuse existing software and data to solve new and more complex problems is extremely relevant and is fundamental for the success of innovative SE approaches. Below, we briefly present two paradigms that may be very useful to realize novel environments for the development of such complex systems.

1.2.1 Multi-agent systems

Intelligent agents and multi-agent systems [86, 66, 39] represent an innovative way of analyzing, designing and implementing complex, heterogeneous and distributed software applications. The agent-based view offers a powerful and high level conceptualization that software engineers can exploit to considerably improve the way in which software is realized. This emerging paradigm permits a high level model of applications to be built, in which many autonomous, intelligent and interacting entities (i.e., a multi-agent system or MAS) interact to share and integrate knowledge, to coordinate themselves in order to achieve a common goal, or to compete in order to satisfy personal needs.

Our interest in agents is driven by the firm belief that they are capable of meeting the demands for complex interaction between the components of an application. We assume a loose "declarative" definition of an agent as an autonomous, social, reactive and proactive piece of software that provides some services and is able to communicate with other agents using a common agent communication language.

The agent-based approach can be very useful to prototype many applications in which legacy software must be integrated into a new distributed system. In fact, it facilitates a compositional approach to software development, that favours the realization of a prototype which performs a global task as the aggregation of sub-tasks, each performed by individual agents. For example, some agents might facilitate access to the functions provided by legacy modules, while others could perform coordination activities among these functions and provide new functions to the external user. The behaviour of coordination components is usually not trivial, and it may depend on many factors, such as incoming information, component's internal state, or the changing of a component's goals as the result of interaction with other entities in the system. The "agent-view" enables to easily program the behaviour of these components.

1.2.2 Logic programming

Logic programming (LP) is a declarative high level programming paradigm that blurs the distinction between specifications and code [83, 46, 80]. Declarative languages have mainly been used in the academic world while the use of imperative paradigms for the development of industrial software has usually been motivated by reasons of efficiency.

There are however some advantages of using LP for SE. LP languages provide a very high level of abstraction and are amenable for formal reasoning. They have better semantics, which makes the detecting and correcting of contradictory knowledge easier. Since LP is declarative, software developers can easily specify *what* an application does instead of programming *how* it performs its functions. A logic program is an executable specification and this encourages rapid prototyping. Besides being reusable, declarative knowledge is more modular and flexible than imperative knowledge and provides abstraction of the (real) world in a natural way. LP encourages the use of meta-programming techniques, that provide support for the integration of different kinds of knowledge and give more flexibility in specifying various policies which drive the behaviour of the application. A logical language can naturally be the target language for the animation of many not executable specification languages [79]. When it comes on the side of integration of heterogeneous systems, logic programming can profitably address semantic integration issues; i.e., the process of specifying methods to resolve conflicts, pool information together and define new operations based on operations belonging to different domains [48].

We believe that LP is a suitable candidate for prototyping innovative agent-based software, mainly because it provides the right mix between formalism and experimentation. It is thus also suitable for developing and validating prototypes of complex applications, where sets of autonomous, intelligent and distributed entities cooperate and coordinate the exchange and integration of knowledge. In fact LP can be profitably used to model the non deterministic evolution of a set of autonomous agents, to dynamically modify the behaviour of agents according to external input or internal modification of their state, to specify rationality and reactiveness of agents, and to represent agents' goals and plans, to represent and structure agent ontologies.

1.3 Goals of the thesis

Agent-based software engineering is a recent and very interesting research area. Due to its novelty, there is still no evidence of well-established practices to the development of agent-based applications and thus experimentation in this direction is very important.

This dissertation presents CaseLP (*Complex Applications Specification Environment based on Logic Programming*), an experimental environment based on logic programming for the rapid prototyping of agent-based software applications. CaseLP provides developers with a prototyping method and a set of tools and languages which allow for the realization of executable prototypes for software applications as societies of interacting agents. The basic idea behind the design of CaseLP was to create an environment which could accommodate various specification and implementation languages as well as legacy software so that a MAS prototype could be built using a range of techniques integrated into a common framework based upon logic programming.

The research on CaseLP has been undertaken with two main objectives. Our first goal was to show that logic programming facilitates the realization of an environment for the (semi) formal development of agent-based prototypes. The second goal was to demonstrate that such an environment can be effectively used for the rapid prototyping of complex real-world software applications. CaseLP especially suits the prototyping of applications involving two issues which today seem to be fundamental: distribution and integration. The former primarily concerns the organization of interactions between the different components that make up the application. The latter is instead related to the integration and reuse of legacy software. In fact, we believe that an essential aspect of a method for the development of new software products is to be open with respect to existing software and its distribution. More precisely, an appropriate method should be able to take into account both new components and legacy systems that will be used in the final applications, and to provide modelling techniques and tools by means of which (partial) specifications and prototypes at various levels of detail can be tested, verified, and refined.

1.4 Overview of the thesis

This dissertation is structured as follows. Chapter 2 first introduces the concepts of agents and multi-agent systems and reviews some application areas where agent technology has been successfully applied. The chapter then presents the main issues related to agent-based software engineering, focusing especially on the development of software applications modelled as MAS as well as on the realization of software architectures for individual intelligent agents. Subsequently, the chapter presents the ARPEGGIO approach, which is the general research framework into which CaseLP is to be inserted. Finally, the chapter briefly presents some tools and methods for the development of agent based software.

Chapter 3 presents CaseLP in details, including the set of prototyping tools that it provides, and the development method that we have defined for the building of MAS based prototypes. The chapter concludes presenting two transport applications that have been prototyped in CaseLP.

Chapter 4 focuses on ontologies in CaseLP, and describes, by means of a running example, how ontologies can be used in our environment to represent knowledge related to specific application domains. Moreover, the chapter presents an ontology that formally defines the features of CaseLP agents.

The ability to realize agent-based software applications that incorporate agents with heterogeneous architectures is very important as it allows agents to be built which optimally perform their tasks. Chapter 5 presents two high-level specification languages and investigates their use for the specification of heterogeneous agent architectures. These languages are the linear logic language $\mathcal{E}_{hhf}$ and HEMASL a simple, imperative, meta-language. The chapter also discusses how these languages could be integrated into the CaseLP framework.

Chapter 6 presents in detail a case-study that has been used as an important testbed for our system. The case-study presented in this chapter concerns the realization of KICKER, a prototype decision support system to be used by European traffic operators in the management of freight train movements between inter-modal centres and yards. KICKER is a project in co-operation between the Information Systems Division of Italian Railways (FS S.p.A.) and the Computer Science Department (DISI) at the University of Genova, Italy.

Finally, Chapter 7 concludes the dissertation summarizing the main research results that have been obtained so far, and discussing the future research directions in which CaseLP is evolving.

Chapter 2

Agent based software engineering

Agents are basically *programming abstractions*, useful for attacking the increasingly complexity of software realization. Agents enhance procedures, modules, and objects, providing a further (higher) level of abstraction that facilitates the development of software applications using a natural view of the system that has to be realized. Agents allow us to analyze, design and implement a software application as a set of interacting and communicating components, which co-operate to achieve a common goal or compete to satisfy personal interests. The agent based view offers a powerful repertoire of techniques, tools, and metaphors that improve the way people conceptualize and implement many types of software.

This chapter introduces first a definition of agent and multi-agent system. It then presents some of the main application areas of agent technology. Section 2.3, focuses on issues related to the engineering of agent software, considering both the development of MAS and of single agents. We present a classification of agent architectures and we describe the most known hybrid architecture, i.e. the BDI architecture. Section 2.4 presents ARPEGGIO, an approach that aims at the realization of an open framework based on logic programming for the rapid prototyping of heterogeneous multi-agent systems. The ARPEGGIO approach is shared among three international LP research groups [21], including the LP Group at the Computer Science Department of the University of Genova. Our research on CaseLP is intended as a contribution to the diffusion of such an approach. Finally, Section 2.5 presents some existing tools for agent-based software engineering.

2.1 A definition of agents and multi-agent systems

Since the scientific community has not yet agreed upon a universally accepted definition[30], a definition of agent and MAS is not easy to give. Nevertheless, the definition that receives the greatest consensus is probably the one given first in [86] and then refined in [39]. According to this definition an agent is

> a computer system, *situated* in some environment, that is capable of *flexible autonomous* action in order to meet its design objectives.

There are three key concepts in this definition: *situatedness*, *autonomy* and *flexibility*.

- Situtedness means that the agent receives input from the environment in which it operates and it can perform actions which change the environment in some way. Examples of environments include the physical environment, the Internet, or a simulated environment. According to the kind of environment, the input may include perception of physical phenomena, message reception from another agent, or interaction with human beings or legacy software modules.
- Autonomy in this context simply means that an agent should be able to act without the intervention of human beings or other agents, and that it should have control over its own actions and internal state.
- Flexibility is a more articulate property and means that an agent is
  - *reactive*: it is able to perceive the environment and respond in a timely fashion to changes that occur in it;
  - *pro-active*: it does not simply act in response to the environment but it is able to exhibit goal-directed behaviour and take the initiative when appropriate;
  - *social*: it is able to interact with other agents and human beings, generally using a sophisticated communication language.

In general, a multi-agent system can be simply defined as

a society of interacting agents.

Agents in a MAS may be either *co-operative* or *competitive*. In a co-operative MAS, the system has a global goal (or set of goals) and the agents that compose the MAS are supposed to co-operate, possibly by performing diverse tasks, in order to achieve the global goal. A typical example is a system that performs distributed problem solving. In a distributed problem solver there is a unique high level goal (corresponding to finding a plan to solve the problem) that is decomposed into parallel activities to be performed by a set of agents. Co-operation and co-ordination among agents are fundamental in performing the overall plan.

In a system where agents are competitive, each of them has its own set of goals, that may or may not agree with other agents' goals. In this case the MAS is an infrastructure that allows agents to interact, each one to pursue personal goals and defend its own interests. A typical example of this kind of MAS is a system for electronic commerce. In such an environment, every agent generally represents either a customer, who wants to buy some goods, or a seller, who wants to sell some goods. Each agent pursues the goals of the person it represents, and these goals are usually in conflict.

In general, there are some other features that characterize a MAS:

- Each agent in a MAS has *incomplete information* about the environment in which it operates and only *limited capabilities* to act within this environment. For example, some agents may have procedural knowledge about how to perform certain subtasks of a complex activity, while other agents may have access to the data on which the procedural knowledge needs to be applied.
- There is *no global control* and every agent has its own autonomy. This does not mean that in general agents are not co-ordinated, but that co-ordination mostly happens via peer-to-peer interaction, instead of using a client-server model.
- Information is *distributed* and not accessible in general by all the agents in the system.
- Computation and communication are *asynchronous*. Each agent has its internal clock and executes independently from other agents. Moreover, agents generally communicate via asynchronous message passing and communication is not intensive.

2.2 Application areas of agent technology

In the last few years agent technology has been increasingly applied in many different fields to solve real-world problems. This section briefly outlines the main areas where agent technology is currently applied (see [39] for more details).

2.2.1 Industrial applications

Industrial applications of agent technology were among the first to be developed and today there are many industrial fields where agent technology is active.

Process control. Process controllers are basically reactive autonomous systems and are an ideal application field for agent technology. Some applications of agents in this field are: a system to control the distribution of electricity [17], a system that monitors and diagnoses faults in nuclear power plants [84], a controller for spacecrafts [38], and a system for climate control [14].

Telecommunications. Telecommunication systems are large and distributed networks of components to be monitored and managed in real-time. Agents have been applied in this field to face problems such as feature integration, network control, transmission and switching, and service and network management. A comprehensive review of this field is presented in [85].

Air traffic control. This is probably the first field where a working agent-based application has been realized. The system is called OASIS [45] and operates at Sydney Airport in Australia.

Transportation systems. Agent technology is well suited for the development of transportation systems. Applications in this field have both a geographically and functionally distributed nature, and are composed by subsystems with a high degree of autonomy and dynamism. For example, traffic management is largely a process of planning, implementation and testing of traffic to optimize the assignment of traffic supplies to traffic demands according to economic and environmental conditions. Of increasing importance is the integration and interconnection of different traffic means (i.e., inter-modal traffic) and their infrastructures. An interesting review on agent-based transportation system is presented in [11]. Chapter 6 of this dissertation presents one inter-modal transport application that has been realized using CaseLP.

2.2.2 Commercial applications

Commercial applications of agent technology tend to be more oriented towards the mass market than towards industrial usage and are certainly the applications that we will more likely encounter in our everyday life. Two main application fields in this area are presented below.

Information management. Information management is basically concerned with the increasing need for selective retrieval of relevant items of information from the enormous amount of information currently available on the Internet. This problem (colloquially known as the *information overload* problem) can be viewed from two complementary perspectives i.e., *information filtering* and *information gathering*. Information filtering studies ways of preventing that a user is presented every day with a considerable amount of useless information (via email, for example) by developing techniques that selectively filter information, retrieving only items relevant for the user. On the other hand, information gathering aims to help a user in finding particular "needles of information" within the ever-increasing haystack of information available on the Internet. A discussion and some applications in this field are presented in [49].

Electronic commerce. Agents are finding a natural application in the automation of commerce, a field which is currently almost entirely controlled by humans. Agent technology provides the flexibility that is needed to realize artificial entities that can operate in marketplaces on behalf of human users, making autonomous decisions, which are strictly dependent on the current context in which the agent is operating and which generally cannot be programmed a priori. Many applications have been realized in this field, for example [26] and [82].

2.2.3 Virtual reality applications

Realization of virtual environments is certainly one of the most promising and challenging fields for agent technology. Agents have an obvious role to play in interactive virtual scenarios, in which human players must interact with "virtual creatures", which have their own "personalities", attitudes and capabilities [27]. Realistic virtual scenarios may be used either in games, as [34] and [67], or in simulations related to civil protection, where virtual scenarios are used to simulate emergency situations, such as earthquakes or flooding.

2.3 Developing agent software

Even though they certainly represent an advance in software engineering, multi-agent systems are generally complex distributed systems, in which each agent has in turn a high degree of complexity. Therefore, it is important to distinguish between the development of software applications as MAS and the development of software artifacts that have (some of) the previously presented features of agents. In the first case, the goal is to realize a software product that will be used in a certain application domain. In the latter case, the software engineer aims at developing a general purpose component, to be used to realize agent-based applications in diverse application contexts.

2.3.1 An iterative approach

For the software engineer that wants to realize an application modelled as a MAS, a software agent is basically viewed as a body of software, which provides some *services* to other agents or human beings by means of interaction with them and the environment in which it is placed. In this case, a two-phase iterative approach can be followed for agent-based software development, before implementing the final application.

1. *Specification of the MAS.* This phase concerns the description of the services that each agent in the MAS provides, the modelling of agent-agent, agent-human and agent-environment interactions, and the description of the domain-dependent procedural knowledge, used by each agent to supply its services and to respond to stimuli from the environment.

2. *Proof of the specification correctness.* The specification properties are formally verified, and/or informal testing of the MAS behaviour is performed by means of a working prototype.

The first phase should not explicitly take into account the *architecture* of each agent but should profitably abstract away from the internal organization and structure of single agents. In this way, the specification is given at the *macro-level*, it is clearer and more modular, and the application developer is not burdened with the modeling of too many details.

On the other hand, the services that an agent provides are usually variegated and complex and generally involve the management of heterogeneous information using different behaviours. In other words, an application needs to incorporate agents employing heterogeneous architectures, so that each application component can optimally perform its task [64]. For example, real-time control systems for industrial machinery could be well implemented using *reactive* agents, whereas a long-term planner of the maintenance activity on all the control systems might be better implemented using a *deliberative* agent[1]. These agents could be incorporated into an application that monitors all the activities of the plant.

In general, it can be very useful to provide the software engineer with a pre-defined library of architectures from which she/he can choose the most appropriate ones for a particular application. Obviously, the architectures in the library must be specified, verified and tested before being used. Specification at the *micro-level* involves the description of internal components of the agent and interactions among them.

2.3.2 Macro-level specification of a MAS

In order to specify an application as a MAS, the software engineer should describe for each agent:

[1]See Section 2.3.4 for an explanation about reactive and deliberative agents.

- the set of *functions* or *services* that the agent supplies to other entities, either other agents or human beings;
- the *input* from the environment, in particular:
  - the *events* the agent can perceive;
  - the *messages* the agent can receive from other agents and/or human users;
- the *output* to the environment, in particular:
  - the *action* the agent can perform;
  - the *messages* the agent can send to other agents or humans;
- the *interaction* the agent has with other entities and the environment;
- the *procedural knowledge* the agent has in order to supply its services and respond to stimuli from the environment.

If a *prototype* of the final application is under development, additional specifications of the environment in which the agents live and its evolution due to agents' actions may be provided.

Services an agent provides represent its public interface towards other entities, either humans or other agents. External entities must be able to *understand*, given the service description, what the agent does exactly when a service is requested. In other words, the agents must share a common intended meaning for the terms that denote services. An agent may reactively provide a service whenever requested (or when a certain event happens in the environment), or may take the initiative and provide a service to another entity pro-actively whenever it decides that the latter requires the service.

The description of the events the agent can perceive depends on the application domain. For example, an agent that controls an industrial chemical plant needs to known the current operation temperature of a chemical reaction, to avoid explosions due to excessive heating. On the other hand, an agent that filters emails that a human user receives, needs to "perceive" the current state of the user's mailbox. In the first case, the agent has to be equipped with sensors that enable it to acquire information from the physical environment. In the second case, the agent should be able to interact with the email demon process running on the user's machine, in order to be notified whenever new messages are put into the mailbox.

If the agent is placed in the physical environment, a description of the set of actions it can perform must be given. These actions will also depend on the application domain. In the case of agent that controls the chemical plant, an action might involve the ignition of a cooling device. In the filter agent case, an action could involve the deletion of messages that are not interesting for the user.

Inter agent and human-agent communication generally happens via message passing. The proper description of input and output messages for each agent is of fundamental importance. An agent must be able to understand the messages it receives, in order to properly provide its services. Therefore, an agent should know, before sending a message, that the addressee will be able to understand it. Agents usually use a *agent communication language* [44] to communicate. These languages generally distinguish between the *performative* of a message and its content. The performative represents the type of communication act being performed, while the content represents the piece of information being transmitted. The receiver is requested to act upon the content of the message differently, depending on the performative of the message. For example, a performative *ask* implies that the receiver is being asked about the truth value of the message content. A performative *inform* implies that the receiver is being informed about something (the message content) that is true for the sender. The differentiation between performative and content of a message implies two levels of description of input and output messages. The first level concerns the description of the type (performative) of messages an agent can send or receive. This level is independent from the application domain of the MAS. The second level of description concerns instead the content of the messages, and is obviously domain dependent. Therefore an agent is able to understand a message if it is able to give a meaning both to the message performative, and to the message content.

Provision of a service takes place by means of an interaction among agents and/or humans. Interactions may range from simple synchronization messaging or the exchanging of query-reply type messages, to more complex forms of negotiation, involving the exchange of many messages. The software engineer should precisely describe how an interaction related to a service will take place.

Agents use their procedural knowledge about the application domain to decide, pro-actively or on demand, whether a service can be provided or not. The software engineer should define, for every agent in the MAS, the proper set of rules and/or plans that compose its procedural knowledge. Procedural knowledge may include mechanisms for co-operation or negotiation. Co-operation mechanisms are used by a group of agents to achieve a goal they have in common. Negotiation, on the other hand, involves the case where two or more agents have to reach an agreement on a certain issue, such as the price for the provision of a service. The procedural knowledge describes both how an agent externally behaves to provide a service and how the internal state of the agent is modified, during the computation, by internal actions.

2.3.3 Micro-level specification of an agent

The internal architecture of an agent is generally defined by several modules which act and interact in various and possibly complex ways. The micro-level specification models the internal components of the agent, as well as the mechanisms which control the activities and the interactions among the various architecture components.

Since in general we want to model MAS with agents using different internal architectures, we need a simple abstraction for agents, which can encompass several of the existing architectures. According to our abstraction, agents at the micro-level are characterized computationally by:

- a *state*, that represents the current image of the agent's computation,
- a *program*, that is a set of rules and/or plans that represent the agent's behavioural patterns (or procedural knowledge), and
- an *engine*, that *organizes* the agent's internal activities and determines the actual agent's behaviour.

The *actual behaviour* of an agent is determined by the actions it takes and it is obtained by applying the agent's program to the agent's state by means of the agent's engine. The *external behaviour* of an agent is determined by the actions it takes that are externally observable, that is physical actions on the environment, or communication actions with other agents. The *internal behaviour* of an agent is determined by the actions it takes that are not externally observable, that is, update actions on the agent internal state.

The state of an agent contains data that may change during its execution. The state typically includes the local *knowledge base* of the agent, that defines the current beliefs of the agent about the environment and other agents in the MAS. The program contains information that does not change during the execution of the agent. An agent is usually programmed in terms of high level rules and/or plans, that give to it the operational expertise it needs to operate in its application domain. In particular, an agent program should define how the agent updates its beliefs and how it acts on the environment and communicates with other agents or humans. The engine controls the execution of the agent and allows the agent to select the right behavioural pattern according to the current state, or to choose the most convenient pattern among the applicable ones.

The engine and the program belong to different abstraction levels: the engine can be considered as a *meta-interpreter* for the program and the data (state) on which the program operates. The actual behaviour of an agent is strongly dependent on how the engine implements meta-level policies. In fact, changing the engine policies means that in the same situation the agent will adopt a different behaviour because it applies a different rule or plan, and this obviously affects the action the agent actually performs.

2.3.4 Agent architectures

We briefly present a classification of the main kinds of agent architectures described in the literature[2]. Our aim is to identify some interesting classes of agents.

In our classification there are four broad families of agent architectures:

- *reactive* agents;
- *pro-active* agents;
- *deliberative* or *rational* agents;
- *hybrid* agents.

Reactive agents behave according to rules which express what has to be done when the agent receives external input. This input can be either a message from another agent or a signal coming from the environment and intercepted by the agent's sensors. Reactive agents do not have a symbolic representation of beliefs or goals and thus do not perform reasoning about them. This does not mean that a reactive agent does not have an internal state, but only that goal-oriented "intelligent" behaviour is an emergent property [10].

Pro-active agents exhibit active behaviour to achieve their goals. Differently from reactive agents their actions are not directly driven by external events (messages or signals), but they can autonomously decide which actions to perform. Also for this class of agents goals remain hard-wired into the behaviour of the agents and are not explicitly represented in their internal state.

Rational or *deliberative* agents are based on a symbolic representation of the world and have an explicit knowledge base, encompassing beliefs and goals. Goal-oriented "intelligent" behaviour is explicitly "coded" into the agents. An agent can usually exploit many different *plans* to achieve the goals that have been ascribed to it. A plan is chosen on the basis of the current beliefs and goals of the agent and can be dynamically modified if the beliefs and/or the goals change. Basically, rational agents can be considered *theorem provers*, that manipulate symbols in order to prove some properties about the world. Implementing an agent as a theorem prover allows us to reuse well known techniques developed in Artificial Intelligence, in particular to realize *expert systems*.

If we look at the agent architectures described above, we immediately realize that none of them defines agents with all the features included in the definition presented in Section 2.1. In particular, they fail in combining real-time responsiveness with goal-directed planning and reasoning. For this reason, research on *hybrid* architectures is very important to build

[2]See [63] for a survey on agent architectures.

agents that can operate in a wide range of situations, facing complex problems in real-time fashion.

The general approach for building hybrid agent architectures defines an agent as a *hierarchy* of interacting layers. Each layer manages a different aspect of the agent's behaviour: lower layers concern reactive behaviour, which the agent performs when it has to quickly respond to external input. More sophisticated behaviours are programmed in the higher layers, that are used by the agent in long term planning of its activity, when rapid reaction to the environment is not needed. In a hybrid architecture the agent's engine controls the interactions between the layers and implements mechanisms to activate and prioritize the activities. In general this is not trivial, since the engine should include very flexible policies, that take into account the great number of situations in which the agent may operate. The engine must decide in each situation which layers to activate and how the activated layers interact, in order that the agent optimally achieves its (implicit or explicit) goals.

Nevertheless, adopting an hybrid architecture gives the following advantages:

- *modularization*: the knowledge base (state and program) of the the agent is profitably partitioned to realize different kinds of behaviours in different layers;
- *robustness*: the same agent can be employed in very different application domains;
- *exploitation of parallelism*: different layers can concurrently work to determine the actual behaviour of the agent.

2.3.4.1 The BDI architecture

The most famous hybrid architecture is certainly the *B(elief) D(esire) I(ntention) architecture* [72]. Figure 2.1 shows the main components that characterize this agent architecture. The BDI architecture agrees with the general abstraction presented in Section 2.3.3 and thus it is characterized by components that contain the state and the program of the agent. Furthermore, it includes the engine that organizes the internal activities of a BDI agent.

The state of a BDI agent is characterized by the following *mental categories*:

- *Beliefs*: the set of beliefs express the representation an agent gives of the state of the world in which it operates. The representation generally includes other agents and may not correspond exactly to the actual state of the world. Beliefs are usually denoted by atomic logical formulas.
- *Desires or goals*: the set of desires or goals specifies an agent's preferences over future world states and its course of action. Goals are generally beliefs, or conjunctions and disjunctions of beliefs, which must be achieved or tested in the current state.

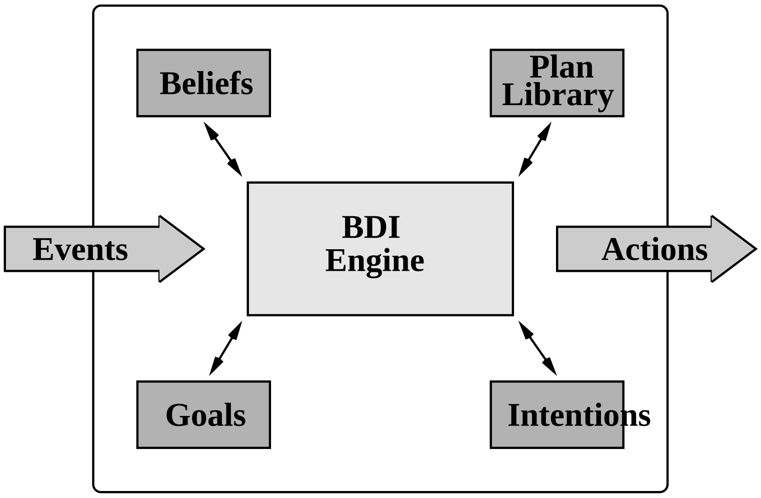

Figure 2.1: BDI architecture

- *Intentions*: the set of intentions contains goals the agent has committed to, together with their state of processing.

The program of a BDI agent is given by a a set of *Plans*, that represent the procedural knowledge of the agent. Plans are important to implement intentions, which are partially instantiated and executed plans. Plans are characterized by:

- a *trigger*;
- a *context*;
- a *body*;
- a *maintenance condition*;
- a sequence of *success actions*;
- a sequence of *failure actions*.

The engine of a BDI agent executes the following cycle:

1. observe the world and the agent's internal state, and update the *event queue* consequently;
2. generate a set of possible new plans, where each plan's trigger event matches an event in the event queue and its context is satisfied;

3. select one from this set of matching plans for execution;

4. push the selected plan onto an existing or new *intention stack*, according to whether or not the event is a (sub)goal;

5. select an intention stack, take the topmost plan and execute the next step of this current plan: if the step is an action, perform it, otherwise, if it is a subgoal, post it on the event queue.

In a plan, the trigger is the event that activates the plan. An event can be perceived by the agent's sensors, be a message received from another agent, correspond to an update in the agent's belief set, or correspond to the execution of a subgoal in a plan. The plan context is a logical condition that has to be satisfied in order that a triggered plan is considered for execution. The plan body is a sequence of atomic actions and subgoals. An atomic action can be an action on the environment, the sending of a message to another agent, or an updating of the agent's beliefs. The execution of a subgoal implies the generation of a corresponding event that is posted in the event queue. When this event is processed, a plan to solve the subgoal is chosen from the agent's procedural knowledge and put into operation. The plan maintenance condition is a logical condition that must remain satisfied while the plan is executed. If the maintenance condition becomes false during plan execution, the plan is immediately interrupted. The sequence of success actions is executed when the plan is successfully completed. The sequence of failure actions is executed if the plan fails (for example when the maintenance condition becomes false). The execution of the failure actions aims to restore the state of the agent to that which it was before the plan started. Of course, it could be the case that not all the actions performed can be reversed, for example it is generally not possible or at least very difficult to undo the effects of communication acts.

A BDI agent is reactively activated by an event perceived from the environment, by a message received from another agent, or by an update which occurs to its knowledge base. Once activated, the agent can rationally decide how to behave, based on which plans are applicable in the current situation. Several policies can be adopted in order to guide the rational (meta) behaviour of the agent. These policies are adopted to select:

- an *event* from the event queue;
- a *plan* among those activated by the event of message;
- a *candidate intention* from the activated plans;
- the intention to be *executed*.

The actual behaviour of a BDI agent determines its degree of reactivity and rationality. The trade off between reactivity and rationality depends on several factors, including the time taken to perform the engine cycle and how sophisticated the policies mentioned above are.

2.4 ARPEGGIO

ARPEGGIO (*Agent based Rapid Prototyping Environment Good for Global Information Organization*) is an approach that advocates the use of logic programming for the realization of an open MAS framework which supports the specification, rapid prototyping and engineering of heterogeneous agent-based software. ARPEGGIO takes contributions from three research groups: the LP Group at the Department of Computer Science at the University of Maryland, USA; the LP and SE group at the Department of Computer Science & Software Engineering at the University of Melbourne, Australia; and the LP Group at the Computer Science Department of the University of Genova, Italy. The basic ideas behind ARPEGGIO have been presented in a position paper [21].

The American partners of ARPEGGIO have been working for years on the *mediation approach to software integration*, developing the mediated system HERMES [48, 2] and the *Interactive Maryland Platform for Agents Collaborating Together* (IMPACT) which builds upon it. An IMPACT agent may be built on top of an arbitrary piece of software, defined in any programming language whatsoever. Each IMPACT agent has an associated *application program interface (API)* that provides a set of functions which may be used to manipulate the data structures managed by the agent in question. The API of a system consists of a set of procedures that enable external access and utilization of the system, without requiring detailed knowledge of system internals such as the data structures and implementation methods used. Each IMPACT agent has an associated *service description* that specifies the set of services offered by the agent.

The Australian partners have developed great expertise in the animation of logic specifications, in order to execute and test them [79], in particular with the PipeDream (*PrototypIng sPEcifcations, Design and REquirements At Melbourne*) project. PipeDream aims to improve the outcomes of requirements analysis by using formal methods, or more precisely mathematical modelling, to determine, analyze and verify requirements. Logic programming can provide the basis for a light weight approach to improved specification analysis. PipeDream analysis involves exploring specifications through animation, and includes limited support form for theorem proving within the logic programming model. The PipeDream approach contrasts with the heavy weight approach of using a general purpose interactive theorem prover, which would require that the developer had detailed knowledge of underlying mathematical theories and proof strategies. While a light weight approach may not give the same levels of assurance as an automated reasoning system, levels of assurance are provided which are adequate for most projects, with significantly less overhead. Animating specifications is particularly promising for developing formal specifications. Animation can be highly automated and thus cheap to perform. Animation can be very effective at detecting problems within a specification because it provides a means of testing a model and its properties *interactively*. These two properties of animation make it very suitable for use early in the software development lifecycle, when the model is more likely to be incorrect or incomplete and still evolving.

CaseLP contributes to ARPEGGIO mainly in the methodological aspects of modelling and implementing the prototype as well as in the integration of different languages for the specification and implementation of an executable prototype.

2.4.1 The **ARPEGGIO** philosophy

We illustrate the philosophy behind ARPEGGIO with a description on what the open framework should consist of. We will assume that we have at our disposal the components depicted in Figure 2.2. At the top of the picture there are some specification languages: each of them is suitable for specifying a particular aspect of the MAS prototype architecture and behaviour. The prototype developers will not necessarily be "experts" in all the specification languages at their disposal but will choose the most useful ones with respect to their needs. For example, there could be a non-executable specification language `A` based on **Z** (a specification language based on first-order logic and set theory) [77] and an executable specification language `eC` based on *linear logic*, which can be respectively used either for modelling the high level interactions between agents or the global evolution of the system. A specification language `D`, based on *algebraic models*, should be suitable for describing the static structure of the agents, while another executable specification language `eB` should help in modelling the dynamic behaviour of a single agent.

The development of a specification is described in the picture by the arrows labelled S. The top layer in Figure 2.2 contains executable specification languages `eA` to `eZ` and non-executable specification languages `A` to `Z`. For any specification language, both executable and non-executable, we assume that there exists a (semi) automatic compiler which can produce an executable form of any specified agent, in a logic programming *target language*. The arrows labelled C on the left of the picture represent this compilation step. If the specification does not describe architectural details for the agents (for example, using *Petri nets* it would be quite easy to describe how the agents interact, but it would be tedious to describe how the agents are structured internally), the (semi-)automatic tool enriches this specification with all the default mandatory details for building the executable prototype.

Once a prototype of the MAS has been created by compiling the various components of

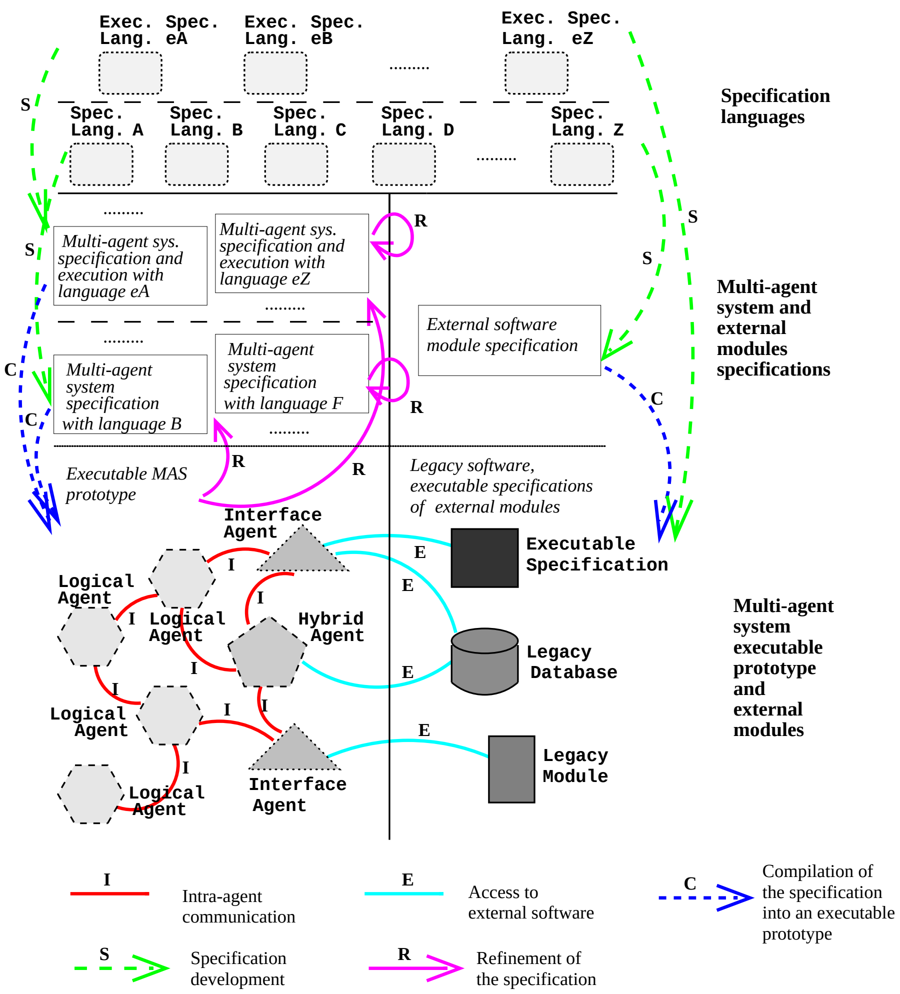

Figure 2.2: The ARPEGGIO open framework.

the specification, we can execute the prototype and so conduct experiments which will detect problems within the specification itself. Animation provides a powerful means of testing a model interactively. Even if it can never prove that the formulation of a model is consistent, or correct, or complete, animation can be used to obtain a similar level of confidence in the consistency, correctness and completeness of a model as that given by program testing, Animation trades off completeness for speed and ease of use.

Besides the other well-known advantages of animation, our approach has two more advantages:

1. We will mix the positive aspects of a formal approach with the satisfaction of needs closer to real world problems: we assume the existence of *interface agents* whose job

is to integrate external software components. All other agents will be specified in a high level language, and when compiled will interact with the interface agents, thus providing an integration of the existing software with the compiled form of the high level specification.

2. We will mix the executable forms of agents specified in different languages: since the *target language* in which the agents are compiled is the same, and since all the details necessary for developing an executable agent are either explicitly specified or added by the compiler, all the compiled agents have an homogeneous form, and are able to interact.

If an executable specification language has been used, it is possible to directly interpret the given specification before compiling and animating it. Both direct execution and animation have the purpose of providing a better understanding of the system, thus facilitating the refinement of the specification, until the obtained executable prototype behaves exactly as expected. In Figure 2.2, the arrows labelled R represent the refinement stage. If the specification language is executable, it is possible to analyze the specification behaviour and to refine it without developing the executable MAS prototype. Whether or not the specification language is executable, a formal approach can be used to verify properties of the specification, and refinement can also be done formally. The looping R arrows describe this refinement cycle. Using compilation will bring us closer to the actual behaviour of the final system because it incorporates the legacy components.

The bottom rightmost part of Figure 2.2 represents legacy software and data that are integrated into the prototype. As well as this legacy and data modules, modules described by external specifications can be taken into consideration. Thus, specification languages can be used not only for defining agents, but also for describing the behaviour of external modules[3]. Using an executable specification language it is possible to develop an executable specification of an external module, which can be treated as any other piece of legacy software. If the language is not executable, an ad-hoc compilation process can produce an animatable form of the specification (see the arrows labelled C in the right part of the picture).

Finally, the *executable MAS prototype* will be implemented in a logic programming *target language*, for example, ECLiPSe [1], Mercury [76], SICStus Prolog [35] or HERMES [81]. At least two types of agents are necessary for accomplishing the complex tasks typical of MAS applications: (1) *logical agents*, which show capabilities of high-level reasoning, carried out through symbolic manipulation and theorem proving, and (2) *interface agents*, which provide an interface towards the external software modules (the access to external software

[3]We distinguish between *agents* and *external modules* since the latter are pieces of software which provide functionality without communication capability, autonomy, or intelligence; they can be inserted into a multi-agent system only if there are agents interfaced with them.

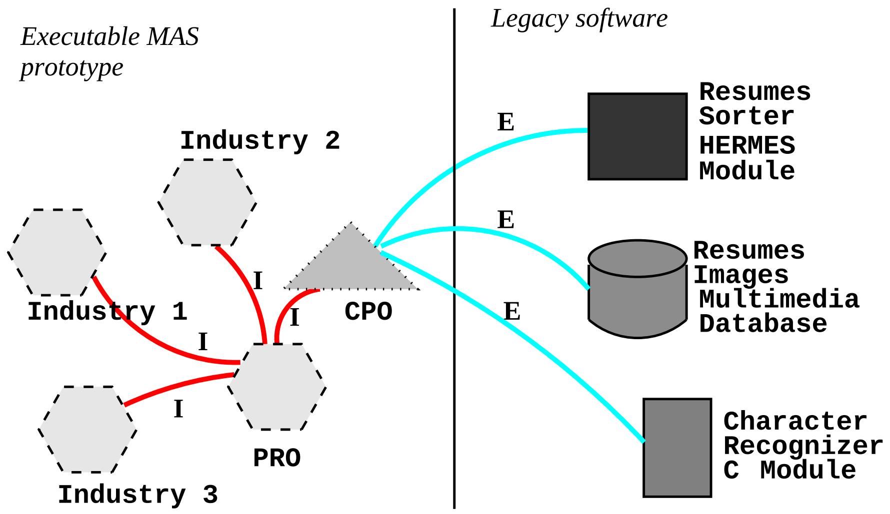

Figure 2.3: The Job-finding agency example.

is represented by the lines labelled E in Figure 2.2). A third kind of agent, *hybrid agents*, combine the skills of the other two types of agents. Communication between agents occurs via message passing in an agent communication language (communication channels are depicted by the lines labelled I in Figure 2.2).

2.4.2 A motivating example

To better understand the nature and roles of the components in Figure 2.2 we have the following example. Consider a hypothetical scenario where the application being modelled involves a *Job-finding* agency (see Figure 2.3), which must cooperate with industries to find jobs for as many candidates as possible. The agency's main components are a *Public Relation Office (PRO)*, modelled by a *logical agent*, and a *Candidate Processing Office (CPO)*, modelled by an *interface agent*. The *PRO* engages in dialog with industries, modelled as logical agents, to find the best solution for all the parties involved. The *CPO* processes the images of the candidates' resumés, contained in an existing *multimedia database*, using an existing C module which recognizes typed characters. The module which indexes the resumés according to their content may use standard text indexing techniques [81], implemented in a full text indexing system accessible through the HERMES mediation system [9, 50]. The interface agent modelling the *CPO* provides the interface with the existing legacy software (*multimedia DB*, *character recognizer* and *Resumés sorter*).

In response to requests from industries, or autonomously at regular time intervals, the *PRO* interacts with the *CPO* to find from the database the resumés of people who match given requirements. Since the *PRO* wants to find jobs for as many people as possible, it tries to combine the industries' requests to optimize the number of engagements. It interacts with industries to propose candidates and adjusts its plans according to the responses from these industries. To do its job, the *PRO* must employ some high-level reasoning mechanism enabling it to behave *rationally*. The *CPO* has the task of interacting with the external modules and integrating their responses before providing a final answer to the *PRO*. Its behaviour is more *reactive*, but the task it performs is complex.

We have started our example by describing the components belonging to the executable MAS prototype, but the execution of the prototype comes only after a specification stage. In the *Job-finding agency* example, this stage would start with the choice of a specification language, continue with the modelling of the interactions between the *CPO*, the *PRO* and the industries, and finish with the animation of the specification to analyze the behaviour of the system. After this stage, it would be possible to refine the specification by choosing another language and modelling lower-level details or aspects different from the ones described previously. The animation of this refined specification will lead to new improvements to the system. After some refinement steps the behaviour of the implemented prototype will match all the initial requirements.

2.5 Tools for agent-based software engineering

This section briefly reviews some existing approaches to agent-based software engineering. We first present Jack [15], a Java-based toolkit for the development of agent-based systems. We then outline three methods for agent-based software engineering: a method for realizing systems of BDI agents [40], the MAS CommonKADS [37], and Tropos [69]. Methods consist of a sequence of steps which enable software engineers to build agent based systems, either at the macro or micro level. A method may, but will not necessarily be supported by automatic software tools to perform the various steps, which are generally only described at a very high level and merely give guidelines for the realization of a MAS application.

2.5.1 JACK

JACK Intelligent Agents by Agent Oriented Software Ltd. [15] is an agent oriented development environment built on top of and fully integrated with the Java programming language. It includes all components of the Java development environment as well as offering specific extensions to implement agent behaviour.

Following the BDI model, JACK intelligent agents are autonomous software components that have explicit goals to achieve or events to handle (desires). Each agent has:

- a set of beliefs about the world (its data set);
- a set of events that it will respond to;
- a set of goals that it may desire to achieve (either at the request of an external agent, as a consequence of an event, or when one or more of its beliefs change); and
- a set of plans that describe how it can handle the goals or events that may arise.

In JACK there are three components:

- The *JACK Agent Language* is the actual programming language used to describe an agent-oriented software system.
- The *JACK Agent Compiler* pre-processes JACK agent language source files and converts them into pure Java. This Java source code can then be compiled into Java virtual machine code to run on the target system.
- The *JACK Agent Kernel* is the runtime engine for programs. It provides a set of classes that give JACK agent language programs their agent-oriented functionality.

2.5.1.1 JACK agent language

The JACK agent language introduces five main class-level constructs. These constructs are:

Agent. The agent construct is used to define the behaviour of an intelligent software agent. This includes capabilities the agent has, the type of messages and events it responds to, and the plans it uses to achieve its goals.

Capability. The capability construct [12] allows for the functional components that make up an agent to be aggregated and reused. A capability can be made up of plans, events, databases and other capabilities that together serve to give an agent certain abilities. An agent can, in turn, be made up of a number of capabilities, each of which has a specific function attributed to it.

Database. The database construct provides a generic relational database. It has been designed specifically so that it can be queried using logical members. Logical members are like normal data members, except that they follow some rules of logic programming[4].

[4] A JACK database can be seen as the extensional definition of a relation in logic programming and queries as non-invertible logic programming goals.

Event. The event construct describes an occurrence that the agent must take action in response to.

Plan. An agent's plans are analogous to functions. They are the instructions the agent follows to try to achieve its goals and handle its designated events.

The JACK agent language is closely related to Java and extends the regular Java syntax. It allows the programmers to develop the components that are necessary to define BDI agents and their behaviour. Each of the events, plans and databases used are implemented as Java classes. They inherit certain fundamental properties from a base class and then extend these base classes to meet their own specific needs. The base classes are defined within the kernel and form the "glue" that holds a JACK agent-oriented program together.

JACK agent language constructs include *declarations* and *reasoning method statements*. Declarations define agent-oriented properties of a JACK agent class, or relationships and dependencies that exist between JACK agent classes. For example, declarations specify which plans an agent use and which events a plan handles. Reasoning statements are JACK specific statements that can only appear in reasoning methods. Reasoning methods represent a peculiarity of JACK which greatly simplify the programming of agents in terms of BDI attitudes. Steps such as posting events, sending messages to other agents or waiting until a particular condition becomes true are all expressed using reasoning method statements.

2.5.2 A method for BDI MAS

In [40], Kinny, Georgeff, and Rao present a modelling technique for BDI MAS that extends object oriented models. The method adopts a double view for a MAS:

- *External viewpoint*: the system is decomposed into agents, characterized by their purpose, their responsibilities, the services they perform, the information they require and maintain, and their external interactions.
- *Internal viewpoint*: the elements required by the BDI architecture (beliefs, goals and plans) are modelled.

External viewpoint. From the point of view of an external observer a MAS is characterized by two models:

- The *Agent model* describes the hierarchical relationship between different abstract and concrete agent classes, and identifies the agent instances which may exist in the system, their multiplicity, and when they come into existence.

- The *Interaction model* describes the responsibilities of an agent class, the services it provides, associated interactions, and control relationships between agent classes. This includes the syntax and semantics of exchanged messages.

In the method, the external models are elaborated as follows:

1. Identify the *roles* in the application domain. Identify *lifetime* of each role. Elaborate an *agent class hierarchy.*
2. For each role, identify associated *responsibilities* and the *services* provided and used to fulfill these responsibilities. *Decompose* agent classes to the service level.
3. For each service, identify the associated *interactions*, the *performatives* required for these interactions and their information content.
4. Perform the *internal modelling* of each agent class, as specified in the following paragraph.
5. *Refine* the agent hierarchy. Introduce *concrete agent classes.* Introduce *agent instances.*

Internal viewpoint. Each of the agents identified in the agent model has a BDI architecture. A BDI agent can be fully specified by: the *events* it can perceive, the *actions* it may perform, the *beliefs* it may hold, the *goals* it may adopt, and the *plans* that give rise to its intentions. In this specification the *execution properties* of the architecture are left implicit. In other words, the development of the internal viewpoint does not include the specification of the BDI engine.

Three models are provided to capture BDI elements in each agent class:

- A *belief model* describes information about the environment and internal state of agents in the class and the actions that agents may perform. The possible beliefs of an agent are described by a *belief set.* One or more *belief states* may define the agent's *initial mental state.*
- A *goal model* describes the goals that an agent may adopt and the events to which it can respond. The goal model consists of a *goal set* and one or more *goal states*, that may define the agent's *initial mental state.*
- A *plan model* describes the plans that an agent may employ to achieve its goals. It consists of a *plan set* which describes individual plans.

The elaboration of internal models starts from *top level goals*, that correspond to the services identified during the elaboration of the external model. Internal models are defined in two steps:

1. *Plan generation*: (a) for each goal, analyze the different *contexts* in which the goal has to be achieved; (b) for each context, *decompose* the goal into *subgoals* and *actions*; (c) generate a *plan* to achieve the goal; *repeat* steps (a) to (c) for subgoals.
2. *Beliefs building*: (a) analyze each required context and conditions for the execution of each and decompose them into beliefs; (b) analyze input and output of each subgoal in a plan and assure that this information is available as beliefs or as outputs of prior subgoals in the plan.

2.5.2.1 The modelling technique

The techniques that are adopted to develop the various models, included in the external and internal viewpoints are based on modelling techniques for object oriented systems.

To develop the agent model and the belief model the method adopts *object models*, that employ object classes and instances to describe different kinds of entities in a MAS at different levels of abstraction. In particular, a set of *class diagrams* is used to specify the agent model. The purpose of class diagrams is to define abstract and concrete agent classes and to capture inheritance and aggregation relationships. In particular, the agent class model defines:

- abstract and concrete agent classes;
- inheritance (also multiple) and aggregation between classes;
- reserved attributes of agent classes;
- beliefs, goals, and plans of each agent class.

Moreover, a set of *instances diagrams* is used to give the agent instance model, that identifies two groups of agent instances: the *static agent set*, instantiated at compile-time, and the *dynamic agent set*, possibly instantiated at run-time.

According to this method, the development of a specification of plans is instead using object oriented *dynamic models*. A plan model consists of a *plan set* in which individual plans are specified as *plans diagrams*. The main part of a plan diagram is the *plan graph*, which is composed of three types of nodes: *start states*, *end states*, and *internal states*, that can be *passive* or *active*. An active node has an associated *activity*: a *subgoal*, an *iteration* construct, or a *subgraph*. Transitions in a plan diagram are labelled with *events*, *conditions* (predicates from the agent's beliefs set), or *actions* (defined in the belief set or pre-defined).

2.5.3 MAS CommonKADS

MAS CommonKADS [37] is a method for agent-based software development. It extends CommonKADS, a method for the engineering of knowledge based systems, with object oriented and protocol engineering techniques. The former are used to give structure to the knowledge that agents have, the latter enable the specification of the interaction and messages exchanged among the agents in the system.

The realization of a MAS is performed building a set of *models*, that capture different aspects of the system to be built.

Agent model. This model expresses the agent's reasoning capabilities, skills, and services.

Task model. This model describes tasks that agents can perform, their goals, task decompositions, and problem solving methods.

Expertise model. This model describes agent knowledge.

Organization model. This model describes the organization in which the MAS is going to be introduced.

Coordination model. This model describes conversations between agents, their interactions, protocols ad required capabilities.

Communication model. This model details the human-agent interactions.

Design model. This model collects the previous models and consists of 3 submodels:

- *network design*: design of network infrastructures;
- *agent design*: composition or decomposition of agents included into the agent model and selection of an architecture for the agents;
- *platform design*: selection of the development platform for each agent architecture.

2.5.3.1 Development method

The development method is based on a *risk-driven* process, that is integrated with a *component-based* approach which allows different parts of the system to be isolated and reused for the construction of other systems. The method is composed of three phases: conceptualization, analysis and design. Below we describe the conceptualization and analysis phases.

Conceptualization. This phase produces the *preliminary description* of the operative scenario in which the MAS will work. *Object Oriented Software Engineering* notation and *Message Sequence Charts* are used to give a set of *use-cases* for some prototyping scenarios of the interactions between the user and the system. Each interaction is given a name and a chart that describes at the high level the messages exchanged between the user and the system.

Analysis. This phase concerns the development of system models, using a risk-driven approach.

In the first step, the agent model is developed and it produces an initial description of system agents. Some strategies for modelling systems in terms of agents are the following:

- Use case actors can be used to delimit external agents. Similar actors can be collapsed into a single agent.
- Syntactic analysis of the problem statement can help to identify agents (the subjects of sentences) goals and services (the verbs of sentences).
- Conceptual distance (knowledge, geographical, logical, or organizational distribution) can be used to identify agents.
- *Internal use cases* can be used. Human-level use cases developed during conceptualization can be decomposed into MAS-level use cases.

In the second step, a top down approach builds the task model as an and/or task tree. The tree represents task decomposition as well as the goals and ingredients of each task. Each task is described by a name, a short description, some input and output ingredients, a task structure, the frequency of task application, some preconditions, and some required capabilities of the performed.

In the third step, the coordination and communication models are given. They describe interactions and coordination protocols between agents and between the human users and the systems. A coordination model formally specifies prototypical interactions between agents and between the system and the environment.

To develop coordination and communication models, the communication channels are first defined according to the following process:

1. describe prototypical scenarios of interactions between agents using Message Sequence Charts notation;
2. model the exchanged messages between agents in each interaction defined above;
3. model each interaction using state transition diagrams of the *Specification and Description Language*;
4. analyze each interaction and determine if it is synchronous, asynchronous or future.

The interactions between the agents are then analyzed, in order to determine if these interactions will require a co-operation protocol. If so, the protocol phases and their combinations must be described.

In the fourth step, the expertise model is developed: *domain knowledge*, *inference knowledge*, and *agents task knowledge* are given. Domain knowledge consists of declarative knowledge of the problem domain, describing *concepts* and their *properties*. Inference knowledge is related to the inference process that is needed to solve a task. Generic inference structures may be adapted in order to execute single steps in the inference process related to a particular task. Task knowledge represents the order of applications inferences structures in task execution.

Finally, the organization model is given. It describe the organization of the enterprise in which the MAS has to be introduced as well as the internal organization of the MAS. This model shows the inheritance relationships between agents using a graphical notation that is based on the Object Model of the *Object Modelling Technique*. Moreover, the notation allows the designer to specify mental state and internal attributes of agents, as well as their external attributes: services, sensors and effectors.

2.5.4 Tropos

Tropos [69] is a method for building agent-based software systems. It is based on two key ideas: the notion of agent and the notion of mentalic attitudes (as beliefs, goals, actions, and plans) that characterize an agent. These ideas are used in all phases of software development. Unlike other methods, Tropos also covers the very early phases of requirements analysis and thus allows the software designer a deeper understanding of the environment in which the software must operate. Tropos targets the realization of MAS which automate processes normally carried out by groups of human within organizations.

Tropos supports five phases of software development:

Early requirements. This phase is concerned with the understanding of a problem by studying existing organizational settings. The output of this phase is an organizational model which includes relevant actors and their respective dependencies. Actors are characterized as having goals which they would be unable to achieve in isolation.

Late requirements. The system under realization is described within its operational environment, along with its relevant functions and qualities. This description models the system as a number of actors which have social dependencies with other actors in their environment.

Architectural design. The system's global architecture is defined in term of subsystems, interconnected by data and control flows. Subsystems are represented as actors, while data and control interconnections correspond to actor dependencies. Actor capabilities and agent types (agents are special kinds of actors) are specified. This phase finishes with the specification of agents within the system.

Detailed design. Each agent of the system is defined in detail, in terms of internal and external events, plans and beliefs, and agent's communication protocols.

Implementation. The actual implementation of the system is carried out, consistently with the detailed design.

2.5.4.1 Early requirements

In this phase, Tropos uses i^*, a model for requirement engineering which provides primitive concepts as actors, goals, and actor dependencies [87]. Early requirements are assumed to involve social actors, who depends on each other for goal to be achieved, tasks to be performed, and resources to be provided. Tropos includes *actor diagrams* for describing the network of social dependencies among actors, as well as *rationale diagrams* for analyzing and trying to fulfill goals through means-end analysis.

An actor diagram is a graph, where each node represents an actor, and a link between two nodes indicates that one actor depends on the other in order to attain some goal. A link is usually labelled with the object of the dependency between the actors. By depending on another actor, an actor is able to achieve goals that it would otherwise be unable to achieve on its own, or not as easily, or not as well. Of course, a depending actor becomes vulnerable: if the dependee fails to provide it with the object of dependency, the actor would negativing affected in its ability to achieve its goals.

The early requirements analysis continues with extending the actor diagram by incrementally adding more specific actor dependencies which are derived by performing a means-end analysis on each of the goal in the diagram. This analysis is specified by rationale diagrams, which are basically obtained exploding goals in the actor diagram. The idea is that goals are decomposed into subgoals and that positive/negative contributions of subgoals to goals are specified.

2.5.4.2 Late requirements

The system-to-be is described within its operating environment. The system is represented as one or more actors which have a number of goal dependencies with the actors in their environment. These dependencies define all functional and non functional requirements for the system-to-be. Some goals are decomposed into internal subgoals of the system and the decomposition is described by "internal" rationale diagrams. Following this analysis the actor that represents the whole system can be decomposed into sub-actors that take on the responsibility of fulfilling one ore more goals.

2.5.4.3 Architectural design

This phase consists of

1. refining the system actor diagram,
2. identifying capabilities and
3. assigning them to agents.

In the first step the system actor diagram is extended according to design patterns that provide solutions for heterogeneous agent communication and the incorporation of non-functional requirements. The second step consists of defining capabilities that actors need to have in order to carry out their goals. A capability is the sets of *events*, *plans*, and *beliefs* necessary for the fulfillment of an actor's goals. The third step consists of defining a set of agent types and assigning capabilities to each. In general, the definition of agents is not unique and depends on the designer.

2.5.4.4 Detailed design

The aim of the detailed design phase is to specify agent capabilities and interactions. Capability specification consists of modelling the external and internal events that trigger plans within the agent as well as the beliefs which the agent use to perform its reasoning. This step involves the use of a subset of the AUML diagrams[5]. In particular:

Capability diagrams. The *AUML activity diagram* allows the designer to model a capability from the point of view of a specific actor. External events define the starting state for a capability diagrams. Within the diagrams, activity nodes model plans, transitions model events, and beliefs are modelled by objects.

Plan diagrams. Each plan node of a capability diagram can be further specified by an *AUML action diagram*.

Agent interaction diagrams. *AUML sequence diagrams* can be used to specify asynchronous message exchanged between agents.

2.5.4.5 Implementation

In general, implementation can be performed using any agent platform. Nevertheless, a BDI architecture naturally supports all the mentalic concepts on which Tropos is based. A possible target platform for implementation is JACK (see Section 2.5.1). In fact, JACK provides the system developer with sophisticated constructs (agent, capability, belief, event, and plan) that find a directed correspondence with the notions previously introduced in detailed design.

[5] AUML is an agent-oriented extension of UML.

Chapter 3

CaseLP: an environment for agent-based software prototyping

CaseLP (*Complex Applications Specification Environment based on Logic Programming*) is an experimental logic-based prototyping environment for complex and distributed software applications modelled as MAS. CaseLP has been under development since 1997 by the Logic Programming Group at the Computer Science Department of the University of Genova, Italy. The basic idea behind the design of CaseLP was to create a software development framework based upon logic programming, which could accommodate various specification and implementation languages as well as legacy software so that a MAS prototype could be built using a range of techniques.

The declarative nature of logic programming languages makes them very suitable for the interactive development and testing of agent software [74, 41, 6]. Logic programming languages can be used to specify agents and MAS at the proper level of abstraction. They can also be executed, thus providing a working prototype "for free" and, thanks to their well-founded semantics, they can be used to formally verify properties of programs, which is fundamental when safety critical applications are developed. The rationale behind CaseLP is to encourage the widespread use of logic programming, and thus attempting to bridge the gap between this technology and specification tools and programming languages traditionally used by software engineers. Moreover, CaseLP tries to simplify the adoption of extended logic programming languages which are most suitable for the formal verification of system properties, but which are not at all user-friendly.

CaseLP is an ongoing research project [54, 55, 56, 57] and thus our prototyping environment is constantly evolving. This chapter presents the current status of the project. CaseLP provides an iterative prototyping method and a set of tools and languages which support the rapid realization of *executable* prototypes of MAS. A prototype can be realized in

CaseLP in a fast and low-cost way, which minimizes the risk of engineering an end product that does not fulfill the user's desiderata. The execution of the prototype allows for early validation of the requirements of a software application in which legacy and new software components must co-operate (in order to provide the user with functionality that goes beyond the functionality offered by any single component).

As stated in Section 2.3, we think that a framework for agent-based software engineering should provide instruments for separately modelling the features of a complete MAS or a single agent. In the current version of CaseLP we assume that the agents have standard architectures and provide the software engineer with an environment for the development of agent software at the macro-level.

The realization of a MAS at the macro-level is performed in CaseLP along three orthogonal dimensions:

- *Task organization* specifies the set of tasks and related services that the prototype has to perform and the set of agents that compose the prototype. It also describes the distribution of tasks and services among agents.
- *Knowledge organization* concerns the knowledge that agents use and manipulate in order to accomplish their tasks. Knowledge may be either declarative or procedural. Declarative knowledge concerns the representation the agents have of the state of world in which they operate. Procedural knowledge concerns rules and plans the agents use to perform the tasks that have been assigned to them.
- *Interaction organization* concerns the way agents interact with each other and how they perceive and act upon their environment.

This chapter illustrates the features of CaseLP. First, we describe its overall architecture and the characteristics of agents we adopt to build up prototypes. Then, we focus on the tools available in CaseLP for engineering a MAS prototype. Section 3.4 describes the prototyping method adopted in CaseLP to engineer a prototype and Section 3.5 presents some applications that have been developed using our prototyping environment. A complete case-study on the application of CaseLP to the management of freight train traffic is presented in Chapter 6.

3.1 CaseLP overall architecture

CaseLP is a system that addresses various aspects of the development of a MAS prototype.

Multi-language modeling. Following the philosophy of ARPEGGIO, our environment provides the software engineer with a set of languages and notations to be used to describe the MAS prototype at various levels of abstraction, from the architectural configuration of the whole MAS prototype to the domain dependent procedural knowledge that agents use to properly behave in their application environment.

Knowledge representation. CaseLP faces the problem of knowledge representation and reuse by exploiting *ontologies* as a means of describing the prototype application domain and the features of CaseLP agents.

Communication. CaseLP allows the software engineer to easily describe communication among agents. It provides a set of tools for specifying high level inter-agent and human-agent communication.

Software integration. CaseLP provides the software engineer with mechanisms to access external software modules. In this way we further both the adoption of a compositional approach for the development of a MAS prototype and the reuse of legacy systems in the realization of new software applications.

Implementation, execution, and simulation. CaseLP prototypes are finally implemented in Prolog. More precisely, there are currently two versions of CaseLP. The first one adopts ECLiPSe [1] as the target language for the implementation of prototypes, and embeds graphical utilities for the interactive execution and off-line tracing of the prototype. The second one adopts SICStus Prolog [35] as the implementation language and contains utilities for executing a prototype starting from different initial configurations and for gathering simulation statistics. In both versions the execution of the MAS is performed by a centralized *scheduler*, realized in Prolog.

Prototyping method. The CaseLP method fully integrates the tools that our environment provides to face the various development aspects stated above. The method guides the software engineer in the development of a MAS prototype, all the way from the informal description of its requirements, to its final implementation.

CaseLP tools and method are described in detail in the next sections.

3.2 Agents in CaseLP

The macro-level description of a CaseLP agent concerns the high-level view of the agent that the software engineer has when she/he wants to realize the prototype of a software application as a MAS. Under this perspective, a CaseLP agent is characterized by:

- a *kind*,
- an *architecture*,
- an *interpreter* and
- a set of *services*.

3.2.1 Kinds of agent

The *Kind* of an agent identifies is as *interface*, *logical*, *facilitator* or *manager*. *Interface agents* allow agents in the MAS prototype to access functions provided by external software systems. They are used to integrate legacy software applications and data into the prototype, which can then be built by combining existing software modules with new components, and this allowing for the integration and reuse of legacy software in the realization of new software products. *Logical agents* use their high level reasoning capabilities to control, co-ordinate, and automate the activities of the components in the prototype. These agents, implement new functionality in the MAS, not provided by pre-existing components.

Example. A software engineer wants to use agent technology to realize a prototype for the efficient execution of multiple queries from several users on a large amount of heterogenous data stored in a pool of sparse databases. To fulfill this aim, she/he could use CaseLP interface agents to integrate the existing distributed databases into the prototype and could use a CaseLP logical agent to implement sophisticated caching or pre-fetching policies that would optimize the global response time for the system. Moreover, CaseLP interface agents could be used to integrate legacy software modules which perform statistical analyses on the use of cached data, in order that the logical agent could decide how to free cached memory.

Facilitator agents supply logical agents with a *yellow-pages service*, that they can use to find agents providing services that they need. An agent may not be aware of all the other agents in the prototype and may need to ask the facilitator to find the services it is looking for. *Manager agents* create and delete other agents in an application. Their services can be used by humans operators, that wish to modify the structure of a MAS prototype during its execution.

3.2.2 Agent architecture

The *Architecture* refers to the internal architecture of an agent. An architecture specifies both the data structures that form the agent's internal components and the flow of control that defines the component activities. CaseLP supports standard architectures for *reactive*, *proactive* and *reactive-proactive* (hybrid) agents[1].

At the micro-level, these architectures conform to the abstraction presented in Section 2.3.3. To summarize, CaseLP agents at the micro-level are characterized by: a state, which represents the current image of the agent's computation; a program, which is a set of rules and/or plans that represent agent's behavioural patterns; and an engine, which organizes the agent's internal activities and determines its actual behaviour. The behaviour of the agent is determined by the application of the agent program to the agent state by means of the agent engine. The architecture of CaseLP standard agents includes a component for the agent program, a component that contain the agent engine, and two components for the agent state. The former contains the current set of the agent beliefs, while the latter is simply a mailbox for incoming messages.

3.2.3 Agent interpreter

In general, external software systems, modules, or data can be written in many programming languages and provide heterogeneous functions. In order to integrate external software into a MAS, CaseLP adopts the *interpretation* approach to software integration [31], that allows the developer to associate a particular interpreter to interface agents. Obviously, the specification of the interpreter depends on the external component that has to be integrated. However, the behaviour of an interface agent follows a the following standard pattern. Whenever an interface agent receives a message that requests a service from the software module which the agent interfaces with, its specialized interpreter is invoked in order to translate the message content into the proper call to the module. Once the external module returns the result, the interpreter translates this result and the interface agent can, in turn, return a message to the requester.

3.2.4 Agent services

Services define functions that the agent provides or requires for accomplishing its set of tasks. Services typically depend on the domain of application of the prototype. Some services can be exported to external human users. Conversely, some services can be imported from outside the MAS and be provided by external software modules.

[1]A classification of agent architectures is presented in Section 2.3.4 of this dissertation.

3.3 CaseLP prototyping tools

In this section we present the tools that CaseLP provides for realizing a MAS prototype. The next section focuses on the CaseLP prototyping method and how the tools are used in performing the different steps that compose the method.

3.3.1 Ontologies

Ontologies are content theories about knowledge domains, developed to clarify knowledge structure and enhance knowledge reuse and standardization. Ontologies allow for the explicit organization of knowledge in agent-based applications, as well as for the unambiguous description of characteristics and properties of agents. Ontologies are used in CaseLP both to express the knowledge about the application domain of the MAS prototype, and to formalize attributes and functionalities of CaseLP agents.

The natural way to model an ontology for an agent-based system based on logic programming is by using a 5-tuple

$$\langle name, terms, predicates, kb, constraints \rangle$$

where *name* identifies the ontology, *terms* is a set of constants and function symbols, *predicates* is a set of predicate symbols to be applied to terms, the knowledge base kb is a set of definite Horn clauses, and finally *constraints* is a set of constraints that kb has to satisfy.

We assume that for each prototype a suitable domain application is explicitly defined. The agent ontology, that describes characteristics and properties of CaseLP agents, does not change from application to application. For this reason the agent ontology does not explicitly appear in the prototyping method. Ontologies and their use in CaseLP are presented in details in Chapter 4 of this dissertation.

3.3.2 MAS-adl

MAS-adl is a simple *architectural description language* for MAS that is adopted in CaseLP for *task organization*, and that is used to describe the prototype's overall architecture in terms of agent templates, instances, their provided and required services, and their communication links.

Architecture description languages have been introduced in *Distributed Software Engineering (DSE)* [42]. DSE addresses the engineering of software that can be distributed and executed concurrently, providing tools and methods for achieving this goal. In particular, DSE researchers argue that a distributed application should be constructed as a *configuration* of interconnected components. To this aim, software engineers should be provided with an architectural description language to express this configuration structure and to make reference to *services* offered at component interfaces, without referring to the actual internal implementation of single components.

Our architectural description language allows for any agent template to specify information about the *kind*, the *architecture*, the *interpreter* and the *services* of the agent. Figure 3.1 presents a schema for the definition of an agent template using MAS-adl and Figure 3.2 shows the grammar that defines its syntax.

agenttemplate ⟨ TemplateName ⟩ {
    kind: ⟨ Kind ⟩;
    architecture: ⟨ ArchType ⟩;
    interpreter: ⟨ IntName ⟩;
    exports: ⟨ ExpSevices ⟩;
    imports: ⟨ ImpServices ⟩;
    provides: ⟨ ProvServices ⟩;
    requires: ⟨ ReqServices ⟩;
}

Figure 3.1: Schema for definition of an agent template in MAS-adl.

MAS-adl describes three main constructs. AgentTemplateDef defines a template for agents used in the MAS under construction. An architectural description generally includes a list of templates. Each template is defined by a kind, an architecture, a possible interpreter for the external application to which an interface agent is connected, and four sets of services. Provided services are services that agents belonging to the template provide to other agents in the MAS. Required services are services that agents belonging to the template request from other agents in the MAS. Exported services represent the interfaces between the prototype and its external human users while imported services represent interfaces to external software modules that are integrated into the prototype.

AgentInstancesDef defines the agents that compose the MAS prototype. In other words, it defines the list of instances of agent templates that are needed to realize the prototype. Multiple instances of a template can be defined.

LinksDef defines directed links between instances of agents. Each link connects a service provided by an agent to a service required by another agent. The communication structure among agents can be set up by the specification of the links. There are three types of link. A one-to-one link connects a provided service to one and only one required service. A one-to-many link is used to connect a provided service to many required services. Conversely, MasArchDescr ::= AgentTemplateDefList AgentInstancesDef LinksDef

AgentTemplateDef ::= **agenttemplate** AgentTemplateName { TemplateDef }

TemplateDef ::= **kind**: Kind; **architecture**: Architecture;
  interpreter: Interpreter; Services;

Kind ::= logical | interface | facilitator | manager

Architecture ::= reactive | proactive | reactive_proactive

Services ::= **exports** ServicesList; **imports** ServicesList;
  provides ServicesList; **requires** ServicesList;

AgentInstancesDef ::= **agentInstances** { AgentInstDefList }

AgentInstDef ::= NumOfInstances AgentName AgentTemplateName

LinksDef ::= **link** { LinkList }

Link ::= OneToOneLink | OneToManyLink | ManyToOneLink

OneToOneLink ::= ReqServ ← ProvServ

OneToManyLink ::= ReqServList ← ProvServ

ManyToOneLink ::= ReqServ ← ProvServList

ReqServ, ProvServ ::= AgentName.Service

AgentTemplateName, AgentName, Service ::= AlphanumericString

NumOfInstances ::= NaturalNumber

Figure 3.2: MAS-adl: a language for MAS architectural description.

a many-to-one link is used to connect many provided services to only one required service. Exported and imported services are implicitly linked to external users and modules and links for them are not defined in the MAS architectural description.

Terms that denote kind, architecture, and interpreter for an agent template are included in the agent ontology, which formalizes the attributes and functionalities of CaseLP agents. Terms that denote services which are included among the four sets of services described above, belong to the domain dependent ontology defined for the application.

3.3.3 KQML

CaseLP agents communicate using point-to-point message passing. The agent communication language adopted in CaseLP is a limited subset of KQML [60, 43]. KQML is probably the most known agent communication language. It is a high-level, message-oriented communication language and protocol for information exchange independent of content syntax and domain ontology.

```
(ask
    :sender joe
    :content price(ibm, PRICE)
    :receiver stock-server
    :reply-with ibm-stock
    :language standard-prolog
    :ontology nyse-ticks)
```

Figure 3.3: Example of KQML message.

Three conceptual levels can be identified in a KQML message: *content*, *communication*, and *message*. The content layer concerns the actual content of the message, expressed in any representation language. The communication layer encodes a set of message features, that describe the low-level communication parameters, such as the identities of the sender and the recipient, and a unique message identifier. The message layer is the core of KQML: this layer determines the kind of interaction one can have with a KQML-speaking agent. The kind of interaction is specified by the *performative* (or speech act) associated to the message. The performative indicates whether the message is an assertion, a query, a command, or any other primitive message type. The performative expresses what the sender of the messages expects the recipient will do when it receives the message. For example, the performative *ask* is used to submit a query to an agent, and the sender of a message of this kind expects that the receiver will formulate a reply to the query.

The syntax of KQML messages is based on a balanced parenthesis list. The first element in this list is the performative of the message, and the following elements are the *arguments* of the performative. An example of a KQML message is shown in Figure 3.3. The sender of the message is an agent called *joe*, that wants to have an answer to the query: *"What is the price of an IBM stock?"*. The recipient of the messages is an agent called *stock-server*. The argument *reply-with* is instantiated with a *conversation identifier*, which the receiver uses in its reply. The content of the message is written in a language called *standard-prolog* and the ontology used by the receiver to interpret the query is identified by the token *nyse-ticks*.

Currently, CaseLP adopts a subset of KQML, that could be expanded if necessary. The CaseLP agent communication language restricts KQML as follows:

- It adopts a limited set of KQML performatives:
  - *ask*: this performative is used by an agent to submit a query to another agent;
  - *reply*: this performative is used by an agent to respond to a query received by another agent;

  - *tell*: this performative is used by an agent to inform another agent about something;

- Since all agents in the MAS are supposed to use the same domain ontology, a message does not contain explicit reference to the ontology used to interpret the message content.

- The communication layer of ask and reply massages does not contain a conversation identifier. Instead, it is embedded into the content of the message, in order that the agent can explicitly manipulate it.

- Since the language we use to implement our agents is based on logic programming, the syntax of the messages is slightly different from the one presented above: messages are terms with the performative as principal functor and the other tokens as arguments. Moreover, the content of a message is always a prolog term.

3.3.4 Tools for conversation specification

Each agent service is provided in a prototype by means of a specific *conversation* between the provider and the requester of the service. Therefore, a conversation has to be established for every service-to-service link defined in the MAS architectural description. A conversation is specified by a sequence

$$m_1, \ldots, m_k$$

of KQML messages. Each message in a conversation is specified by:

- the *performative*;

- the *content*;

- the *sender*;

- the *receiver*.

Given the performatives included in the CaseLP agent communication language, there are two typical conversations:

- an agent requests a service from another agent, which replies reactively to the request, providing the requester with the information it needs;

- an agent proactively provides a service to other agents, and does not require any reply from them.

Two messages are exchanged in the first type of conversation. The first is an *ask* message from the requester to the provider. The second is a *reply* message from the provider to the requester. Just one message is exchanged in the second type of conversation, in this case a *tell* message is sent from the provider to the requester of the service.

In general, it may be the case that some conversations are started by other conversations[2]. Such situations are properly captured by introducing the concept of a *sub-conversation* and by providing a notation for expressing which conversations possibly start during other conversations.

Let $c_1 = \{m_1, \ldots, m_k\}$ be a conversation composed by messages $m_1, \ldots, m_k$ and let c_2 be another conversation. The notation

$$c_1 sc_{m_i} c_2$$

means that c_2 must start after message m_i in c_1 has been handled by the receiver. The notation

$$c_1 sc?_{m_i} c_2$$

means that c_2 may start after message m_i has been handled by the receiver. In this case, the decision about starting c_2 is up to the receiving agent and depends on its internal state.

The *conversation model* of the MAS includes the definition of all conversations that take place for the provision of services in the prototype, as well as the specification of sub-conversation relationships between conversations, which are described using the notation given above.

3.3.5 AgentRules

AgentRules is a rule-based agent programming language that is used to define the initial state of agents and to specify their program. AgentRules is used to program reactive, proactive and reactive-proactive agents. It is very similar to ProlAg, the logic programming language that is used in CaseLP for prototype implementation (see Section 3.3.6) and just adds some syntactic sugar to it, in order to provide prototype developers with a more user-friendly language. Actually, AgentRules programs have to be translated into ProlAg before prototype execution.

The grammar that defines the syntax of AgentRules is presented in Figure 3.4. An agent is given an *initial state* which is a (possibly empty) set of atomic ground formulas. The initial state expresses the agent's initial knowledge about its application domain.

An agent program is a set of reactive and/or proactive rules, which define its procedural

[2]For example, imagine the case where an agent a requires a service from another agent b, that must require an accessory service from another agent c, in order to supply the service requested by a.

InitialState ::= nil | BeliefsList
BeliefsList ::= Belief | Belief; BeliefsList
Program ::= **program** ReactiveRulesList ProactiveRulesList **endprogram**
ReactiveRulesList ::= nil | ReactiveRule | ReactiveRule; ReactiveRulesList
ReactiveRule ::= **on message** Msg **check** Condition **do** ActionsList
ProactiveRulesList ::= nil | ProactiveRule | ProactiveRule; ProactiveRulesList
Msg ::= Performative { **content**: Content; **sender**: Sender; **receiver**:Receiver; }
Condition ::= StateCondition **and** AuxiliaryCondition
StateCondition ::= true | Goal
AuxiliaryCondition ::= true | Goal
Goal ::= A | A **and** Goal | A **or** Goal
ProactiveRule ::= **check** Condition **do** ActionsList
Action ::= assert_state(Belief) | retract_state(Belief) | send(Msg, Receiver)

Figure 3.4: AgentRules: a language for agent state and program specification.

knowledge. The program of a reactive agent is expressed by means of *event-condition-actions* rules. An event-condition-actions (reactive) rule has the following form:

on message Msg **check** Condition **do** ActionsList

In order to execute the sequence of atomic actions ActionsList that compose the body of the rule, the reactive rule has to be triggered by a particular message Msg, received from another agent, and at the same time, the condition Condition, associated to the rule, has to be satisfied within the agent state.

Condition is actually the conjunction of two distinct conditions. The first is a constraint on the possible agent states, and expresses a logical condition that the set of beliefs that compose the agent state must satisfy. The second is an auxiliary condition, and is actually made up of a set of calls to auxiliary procedures. If all these calls succeed, the auxiliary condition is satisfied. An auxiliary procedure is either a prolog-like clause defined in the agent code, or a prolog built-in procedure. State and auxiliary conditions are called by invoking goals. A Goal is either an atomic formula A, or a conjunction or disjunction of atomic formulas.

Atomic actions are either updates to the agent state or the sending of messages to other agents in the MAS. The argument of the actions *assert_state* and *retract_state* is a belief that is added to or removed from the agent's state respectively. The arguments of the action *send* are a message and the name of the receiving agent.

Proactive agents behave accordingly to *condition-actions* (proactive) rules. A condition-action rule has the form

check Condition **do** ActionsList

In order to execute the sequence of atomic actions ActionsList, the condition Condition has to be satisfied within the agent state.

Programs for reactive (proactive) agents are composed only by reactive (proactive) rules. Programs for reactive-proactive agents are obtained by mixing proactive and reactive rules.

The AgentRules code that defines the program for an interface agent has to contain at least one rule that activates the agent interpreter, which provides access to the external software to which the agent is linked. Interpretation rules have the form

on message
`ask(content:  X, sender:  Sender, receiver:  Receiver)`
check
`interpret(X, Answer)`
do
ActionsList

where ActionsList contains a send action of the form

```
send(reply(content:  Answer, sender:  Receiver, receiver:  Sender))
```

The auxiliary predicate *interpret* implements the real interface between the agent and the external model. The interface maps the message content X into a proper call to the external module, collects the result, and maps it back into *Answer*.

3.3.6 ProlAg

Once all the agents in the prototype have been specified using the languages and tools described above, an executable prototype must be implemented in order to run a simulation. In particular, for each agent in the system, we must implement its initial state, its program and its engine. Moreover, for interface agents, the code that defines the agent interpreter must be provided.

The CaseLP environment initially used ECLiPSe as the implementation language. Subsequently, a new version of CaseLP adopted SICStus Prolog. In order to be independent form the typical features of these logic programming language, we present a platform-independent logical language for the implementation of the agents that are included in a prototype.

ProlAg(*Prolog for Agents*) is the target language for the implementation of the MAS prototype. ProlAg extends standard prolog with primitives for *safe* state update and for communication. It is used to implement the state, program, and engine of agents that are incorporated into the prototype. ProlAg can also be used for the direct implementation of the agent code, if the developer is an expert in logic programming.

In ProlAg there are two predicates that allow an agent to update its state.

- *assert_state(Fact)* asserts the fact *Fact* into the agent state. Its implementation is safe since, if the predicate is backtracked, *Fact* will be removed from the state.
- *retract_state(Fact)* removes the first fact in the agent's state that unifies with *Fact*. If it is backtracked, *Fact* is re-asserted.

One predicate allow an agent to send a message.

- *send(Receiver, Message)* sends *Message* from the calling agent to the agent *Receiver*.

Three predicates allow an agent to read messages.

- *sync_receive(Message)* blocks the agent until a message unifying with *Message* enters the mail-box.
- *async_receive_one(Msg_input, Msg_output)* searches the agent's mail-box for a message unifying with *Msg_input*. If such a message is present, *Msg_output* is unified with it, otherwise *Msg_output* unifies with the atom *no_message*.
- *async_receive_all(List_of_messages)* fetches all the messages in the mail-box of the agent. These messages are unified with *List_of_messages*.

Example: implementation of CaseLP standard agents. Due to their rule-based nature, AgentRules specifications can be easily translated into a form that can be manipulated by a reactive-proactive engine. We outline how the AgentRules specification of a reactive, proactive, or reactive-proactive CaseLP agent can be implemented in ProlAg. Moreover, we discuss how ProlAg can be used to implement an engine for a reactive-proactive agent.

The implementation of an AgentRules specification in ProlAg conforms the schema presented in Figure 3.5. The state of an agent is specified in AgentRules as a set of ground

```
% % % % Initial state % % % %
fact(B₁).
...
fact(Bₙ).

% % % % Program % % % %
reactive_rule(Message₁, Condition₁, Actions₁).
...
reactive_rule(Messageₘ, Conditionₘ, Actionsₘ).
proactive_rule(Conditionₘ₊₁, Actionsₘ₊₁).
...
proactive_rule(Conditionₘ₊ₖ, Actionsₘ₊ₖ).

% % % % Auxiliary predicates % % % %
p₁(..):- ...
...
pₛ(..):- ...
```

Figure 3.5: Schema of ProlAg implementation of a AgentRules specification.

facts $\{B_1, \ldots, B_n\}$, that represent the agent's beliefs. It is straightforward to map this AgentRules specification of an agent state into ProlAg atomic ground formulas of the form `fact(B`$_i$`)` $(1 \leq i \leq n)$.

The program of an agent is defined by two sets of ProlAg clauses. The former corresponds to the set of reactive, proactive, or reactive-proactive AgentRules clauses. A reactive rule

on message Message **check** Condition **do** Actions

is mapped into a ProlAg clause with the following schema

`reactive_rule(Message, Condition, Actions)`,

where *Message* is a term that denotes the triggering message, *Condition* is a term that denotes the conjunction of conditions on the agent state and auxiliary conditions, and finally *Actions* denotes the sequence of AgentRules atomic actions.

Analogously, it is straightforward to define the mapping from proactive AgentRules rules into ProlAg. A reactive rule

check Condition **do** Actions

```
cycle :-
  async_receive_one(Msg_input, Msg_output),
  (Msg_output == no_message ->
    Actions1 = []
  ;
    select_reactive_rule(Msg_output, Actions1)
  ),
  select_proactive_rule(Actions2),
  choose(Actions1, Actions2, Actions),
  execute(Actions).
```

Figure 3.6: Engine for a reactive-proactive agent in ProlAg.

is mapped into a ProlAg clause with the following schema

```
proactive_rule(Condition, Actions),
```

where *Condition* is a term that denotes the conjunction of conditions on the agent state and auxiliary conditions; and where *Actions* denotes the sequence of AgentRules atomic actions.

The second set of clauses $\mathtt{p}_1$, ..., $\mathtt{p}_s$ that are included in a ProlAg program corresponds to the set of AgentRules auxiliary procedures that are used to verify the auxiliary conditions mentioned above. If we consider an interface agent, one clause $\mathtt{p}_j$ will assume the form

```
interpret(X, Answer) :- ...,
```

and implements the agent interpreter.

The engine for a reactive-proactive agent can be implemented in ProlAg as shown in Figure 3.6. The predicate `cycle` defines a simple engine (meta-interpreter) for the agent architecture. Cycle executyion proceeds as follows. First, a message is received asynchronously. Then, a fired reactive rule is selected from the agent program and the set of associated actions is returned. Selection among proactive rules is also performed, and a set of proactive actions is returned. Finally, a set of actions is chosen and executed. Obviously, more sophisticated strategies for rules selection and action choice can be taken into account in a more complete engine. Moreover, other communication protocols may be implemented. For example, synchronous communication can be implemented using the available variants of the primitive *receive*.

3.3.7 Execution tools

After all the agents have been implemented as ProlAg pieces of code, they can be compiled into the target logical language (ECLiPSe or SICStus Prolog). Compilation produces code for the MAS, which is then ready to be simulated using the CaseLP run-time support. The software engineer has two possibile ways of running the simulation. The first way is to use the CaseLP Visualizer, a graphical tool that provides the user with a friendly GUI to load the agent code into the system, start the simulation, and follow the traceable events while the simulation is running and after its conclusion. At the moment, the CaseLP Visualizer is available only in the ECLiPSe version of CaseLP. The second way to run the simulation is by issuing commands from within the CaseLP Shell, which is available both for SICStus Prolog and ECLiPSe. The CaseLP Shell available under SICStus Prolog allows the user to parameterize some features of the simulation, in order to test the prototype in various application scenarios.

The execution of the prototype is based on a *round-robin scheduler*, which activates in turn all the agents in the MAS. When activated, an agent executes the cycle that implements its engine and behaves according to the rules defined in its program. When the agent finishes the execution of its cycle, the scheduler activates the next agent. The CaseLP scheduler controls the global clock for the MAS execution. Once all the agents have been activated in succession, the simulation time is increased by one time unity, and another round is started. This process continues until a given number of iterations is reached. The CaseLP scheduler also manages the delivery of messages. When a currently executing agent sends a message, the scheduler puts the message into the mail-box of the receiver, in order that it can properly receive the message when it is activated.

3.3.7.1 CaseLP Visualizer

The CaseLP Visualizer [68] provides the software engineer with a user-friendly GUI, which can be used to load agents into the system, start the execution of a prototype, and trace the events that take place during simulation.

In order to execute the MAS, the system user initializes the messages that constitute the input of the simulation (and that correspond to requests for services exported by agents from within the prototype). The CaseLP Visualizer provides an appropriate window for performing this task, as illustrated in Figure 3.7.

The user can also set the duration of the simulation (expressed in cycles of the MAS global clock) and the granularity of on-line visualization. In this way the simulation will periodically stop to display the events that have occurred.

The visualizer allows both on-line and off-line visualization of the MAS execution, focusing

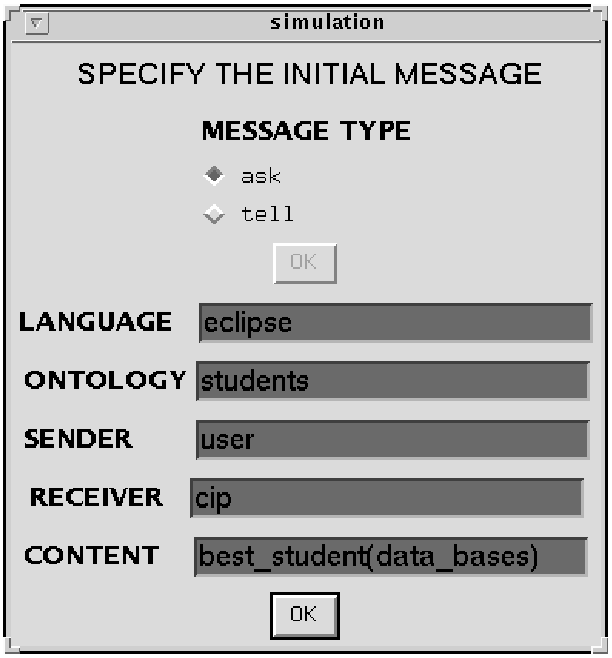

Figure 3.7: CaseLP Visualizer: initialization window.

on those events the user is most interested in. Three kinds of events can be displayed: agent state updates, message sending, and message reception. According to the developer needs, the code of the agents is automatically *instrumented*. Instrumentation is the process whereby probes are added to the agent code, in order to collect and store for subsequent display the events that occur during execution of the MAS.

While the simulation is running, windows for each loaded agent appear on the screen. Information about state changes and exchanged messages is displayed for each agent. Figure 3.8 presents a snapshot of a MAS execution. After the execution has ended, it is possible to view a more detailed trace of the events which have occurred, as illustrated in Figure 3.9. Message exchanges and state updates are shown for each agent. By clicking on an event, it is possible to see more details about the event, as shown in Figure 3.10. Both the on-line and off-line visualization modalities provided by the CaseLP Visualizer are useful to monitor the overall behaviour of the prototype, in order to check that it behaves correctly.

3.3.7.2 CaseLP Shell

Instead of running a simulation using the CaseLP Visualizer, the prototype developer can also use the CaseLP Shell, and type in commands to execute the simulation.

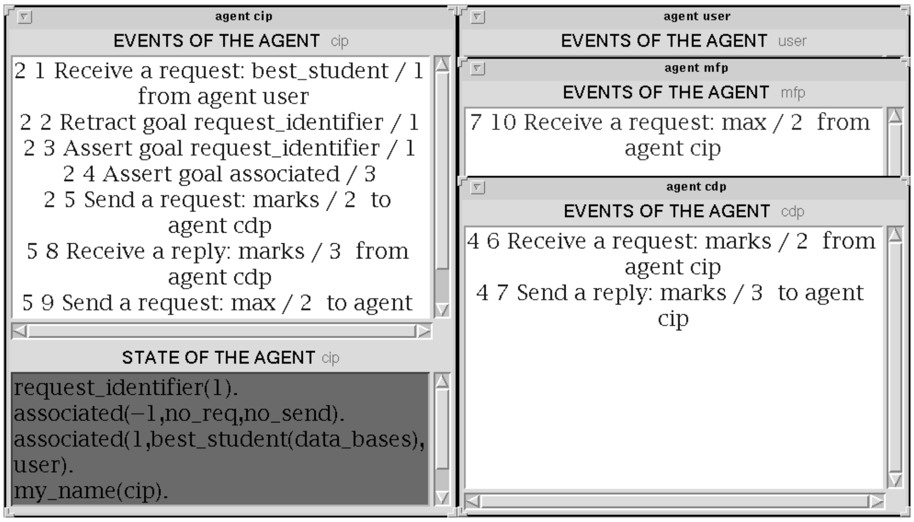

Figure 3.8: CaseLP Visualizer: on-line visualization of execution.

- The command `compile` allows the user to translate an agent written in ProlAg into the target logic language (ECLiPSe or SICStus Prolog).
- The command `load(File_name,Agent_name)` loads a previously compiled agent from a file into the system.
- The command `initialize(List_of_initial_messages)` initializes the input messages for each agent.
- The command

  ```
  global_parameters(Iter, Min_time_unit, Out, Verbose)
  ```

  sets the global simulation parameters. The parameter *Iter* represents the number of iterations (scheduler cycles) over which the simulation will be performed, expressed in terms of the unit *Min_time_unit*. The parameter *Out* is the name of the file into which copies of all exchanged messages will be placed,. The parameter *Verbose* is a flag signaling whether or not the messages will be displayed on the screen during the simulation run.
- The command

  ```
  net_parameters(From_ag,To_ag,Failure_rate,Min_latency,
                 Max_latency,Unit)
  ```

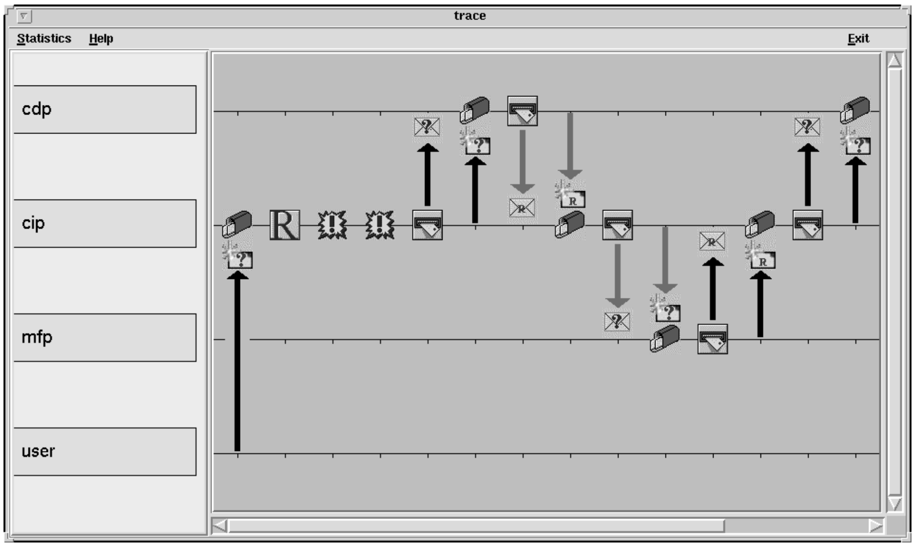

Figure 3.9: CaseLP Visualizer: off-line tracing of execution.

allows the user to define, for any pair of interacting agents, the failure rate and the minimum and maximum latency of their communication channel. The parameters *Min_latency* and *Max_latency* are the minimum and maximum values of the delay over the communication channel and are expressed in terms of the parameter *Unit*. The arrival time of a message, that is set by the CaseLP scheduler, will have a delay in the range of *Min_latency* to *Max_latency* from the time of posting. If the parameter `net_parameters` is not defined for a pair of communicating agents, then the channel between them is assumed to be without delay and to be completely reliable.

- The command `start_simulation` starts the execution of the prototype, respecting the parameters described above.

- The command `get_statistics` can be used to collect statistics about the simulation after it has finished. The statistics output file contains, for each simulation interval, the number of messages received in the interval and the sum of their weights[3]. The total number of exchanged messages, the maximum and average number of messages per interval, the maximum and average weight per interval, and the standard deviation of message weight are also provided. In some experiments this information has proved useful for evaluating the occupation of communication channels, which is of fundamental importance in applications where communication bandwidth is limited.

[3]Each message is given a *weight*, that represents some measurable property of the message itself. For example, the weight of a message can represent the bandwidth that is occupied when the message is transmitted.

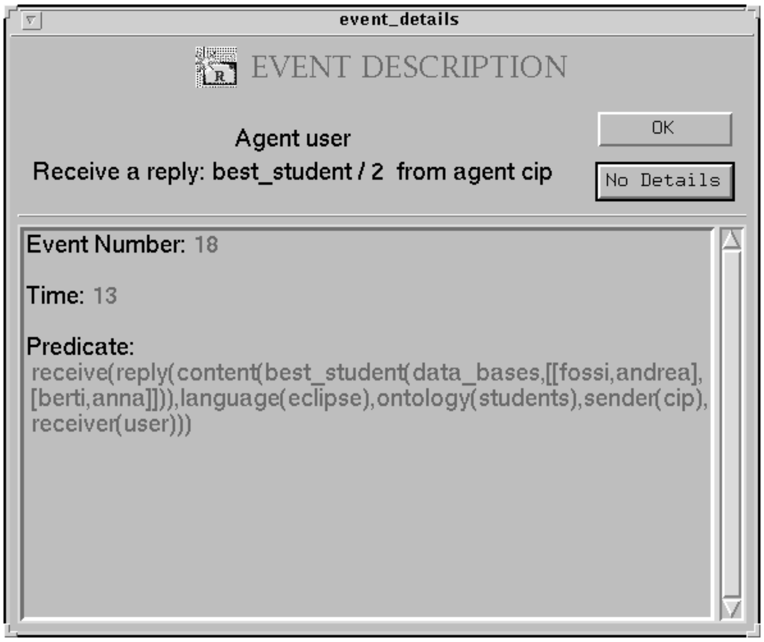

Figure 3.10: CaseLP Visualizer: details of an event.

3.4 CaseLP prototyping method

CaseLP provides agent-based software engineers with a development method that enables them to realize an executable prototype of a software application as a society of interacting agents. The CaseLP development method is based on an iterative refinement of the prototype. Each time a release of the prototype is realized, its global behaviour is tested and validated using animation. Interaction with the costumer in this phase is fundamental for validating the correct functioning of the prototype. According to the outcome of testing and validation, a new version of the prototype may be realized and possibly further refined until the development process converges upon the customer's desiderata. The prototyping method includes both informal and formal phases and tries to conjugate rigor with simplicity of use.

Since its aim is to develop agent prototypes at the macro-level, the method mainly takes into consideration the three macro-level dimensions of agent development previously presented, which were *task organization*, *knowledge organization*, and *interaction organization*. A MAS-based software prototype is realized in CaseLP according to the development method represented graphically in Figure 3.11. The method develops a prototype at the macro-level and assumes that all the agents in the MAS have a standard CaseLP architecture. In the following, we present all the steps in this method. We distinguish three

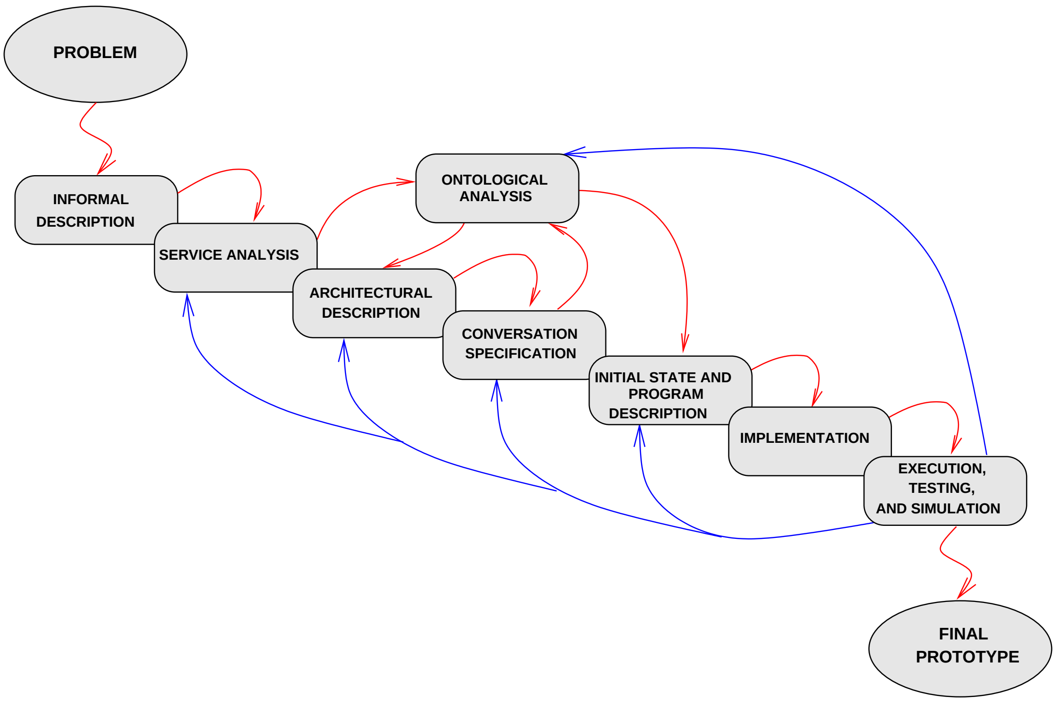

Figure 3.11: The CaseLP prototyping method.

separate phases in the method adopted by CaseLP. Steps 1 through 6 form the *modeling phase* of the prototype. Step 7 represents the *implementation phase* and Step 8 represents the *execution phase*.

3.4.1 Step 1: Informal description of the features of the application

In the first step of the CaseLP method, an informal description of the prototype is given. This description takes into consideration both *task organization* and *knowledge organization*.

A functional description of the prototype is given in natural language. The description consists of the high level tasks that the prototype has to perform. Some tasks may be predefined because they are carried out by legacy software applications which are to be included in the prototype. Other tasks may be new and could, for example, provide coordination functionality between legacy modules, or perhaps high level reasoning in order to perform critical optimizations.

A description of the prototype based on knowledge organization is also given in natural language. The description consists of the information that the prototype has to manipulate in order to perform its functions. If the prototype is built upon legacy software applications, the description includes the input/output information they manipulate.

Note that in this and the next two steps we do not explicitly introduce agents and other architectural details of the prototype. This is because we wish to remain at an abstraction level that does non commit to any choices about implementation and to concentrate on the functional and information aspects of the prototype.

3.4.2 Step 2: service analysis

This step strictly relates to *task organization*. The tasks informally identified in the previous step are formalized as a set S of *service names*. Services represent the "operational" means by which a task is performed. Each task is performed by accessing a set of services included in S. The set S is actually partitioned into three subsets. The first set S^e contains the services that are exported from the prototype to external (human) users. The second set S^{imp} contains the services that are imported into the prototype and are provided by external legacy software modules.

The last set S^{int} contains the services that are used internally by the prototype. In other words, S^{int} is the set of services that some agents in the MAS provides, while other agents in the MAS request.

3.4.3 Step 3: ontological analysis

This step strictly relates to *knowledge organization* and concerns the formal definition of the information manipulated by the prototype. According to the informal description obtained in Step 1, the prototype developer decides upon the domain ontology to be used. The domain ontology is completely independent of the MAS-based prototype in which it is being used. Therefore, it may be the case that there is an already existing domain ontology that can be used or adapted, for use in the current application. If this is not the case, an ad hoc domain ontology for the application must be realized. This involves the setting of each element of the 5-tuple which identifies an ontology.

Name. The name of the ontology is chosen.

Terms. The relevant objects of the application domain are given a representation by means of a set of terms that are included in the ontology language.

Preds. The relevant properties of the objects of the application domain and their relationships are given suitable names by means of a set of predicates. These predicate names are included in the ontology language.

Kb. The knowledge base of the domain ontology is built by defining logical properties and relationships on terms using the ontology predicates.

Constr. The constraints on the knowledge base are defined by means of suitable predicates in the ontology language.

As well as defining the domain ontology, this step also partially defines the domain dependent part of the ontology *agent* (see Section 4.3.2), that structures the knowledge about agents related to the particular application domain in which they operate. To this aim, the ontology *agent* includes the predicate $domain\_service/2$ to define domain level services. A term s and a predicate $domain\_service(s)$ are added to *agent* for each domain service $s \in S$ selected in Step 2.

3.4.4 Step 4: prototype architectural description

In this step the developer uses MAS-adl to define the static structure of the MAS which models the prototype. Services identified in Step 3 guide the definition of the agents that compose the prototype, both in terms of their internal characteristics and of the communication channels which exist between them. This step consists of:

1. Choosing the agent *templates* required by the application. Each agent template at is characterized by:
   - a *kind* k for the agent, among the ones available in CaseLP;
   - an *architecture* $arch$, that determines the internal structure and the functioning of the agent;
   - an optional *interpreter* int, used to access the legacy software attached to the agent;
   - a set of services $S^{+}_{at} \subseteq S^{int}$, *provided* by the agent in order to perform its tasks;
   - a set of services $S^{-}_{at} \subseteq S^{int}$, *requested* by the agent in order to perform its tasks;
   - an optional set of services $S^{e}_{at} \subseteq S^{e}$, *exported* by the agent to external (human) users of the prototype.

- an optional set of services $S_{at}^{imp} \subseteq S^{imp}$, *imported* by the agent from software modules external to the prototype.

2. Selecting the set I_{at} of *necessary instances* for each template at.

3. Setting the *interconnections* between the instances of agents, linking appropriately requested and provided services. The set of links L will contain pairs of the form $(a(s^+), a'(s^-))$, where a and a' are agent instances and s^+ and s^- are the services provided by a and requested by a', respectively. Note that in the denotation of a link we use the same service name s.

3.4.5 Step 5: conversation specification

This step strictly relates to *interaction organization* and concerns the definition of *conversations* that are used to provide services within the prototype. We assume that a conversation is a sequence of messages and consider two types of conversations: *internal* and *external*. An internal conversation occurs between two agent instances that provide and request an internal service. An external conversation take place either between a (human) user and an agent in the prototype, or between an agent in the prototype and an external software module. An internal conversation is set for each link $l = (a(s^+), a'(s^-))$ defined in Step 4. An external conversation is set for each service exported or imported by any agent instance.

Since a conversation is independent of the content of the messages that compose the conversation, we first decide upon the content of the messages that are exchanged. For each link l, a corresponding set C_l of message contents is chosen. Each $c \in C = \cup_{l \in L} C_l$ is added as a term in the *agent* ontology and a corresponding predicate $domain\_message(c)$ is inserted into the knowledge base of *agent*. Moreover, a predicate $domain\_associated\_to(c, l)$ is added to the knowledge base of *agent* to express that the particular message content c is associated with link l.

After the *agent* ontology has been filled with all the message contents and related predicates, the actual conversation is decided upon for each link l. This is done as follows.

1. A set of messages $M_l = \{m_1, \dots, m_k\}$ is chosen for inclusion in the conversation. For each message $m \in M$ a performative is chosen. A message content is also chosen from those associated to the link l in the *agent* ontology.

2. The conversation $c_l = m_{i_1}; \dots ; m_{i_k}$ is set, as a sequence of all the elements of M_l.

Once a conversation has been defined for each communication link, sub-conversations have to be defined using the notation presented in Section 3.3.4.

3.4.6 Step 6: initial state and program description

This step uses AgentRules to set up the initial state and program for each of the agents in the prototype. The initial state of each agent instance is given by a set of facts that represents its initial knowledge about the application domain. The specification of agent programs is based on the conversation model defined in Step 5:

- For each conversation *conv* between a pair of agents a and b:
  - For each message $m \in conv$ from a to b:
    - define $k \geq 1$ reactive rules $rr_{m_s}(1 \leq s \leq k)$ to be inserted in the program of b. Each rule rr_{m_s} is triggered by message m.
    - define $h \geq 0$ proactive rules $pr_{m_s}(0 \leq s \leq h)$ to be inserted in the program of a.
  - For each message $m' \in conv$ from b to a:
    - define $k' \geq 1$ reactive rules $rr_{m'_s}(1 \leq s \leq k')$ to be inserted in the program of a. Each rule $rr_{m'_s}$ is triggered by message m'.
    - define $h' \geq 0$ proactive rules $pr_{m'_s}(0 \leq s \leq h')$ to be inserted in the program of b.
- For each reactive rule rr of an agent a, set the state condition, the auxiliary condition, and the set of actions of the rule as follows. Now consider the message $m \in c$ that triggers the rule:
  - if $csc_m c'$ then set both the conditions to *true* and include the sending of the first message m' of c' in the set of actions;
  - if $csc?_m c'$ then include the sending of the first message m' of c' in the set of actions.
- For each proactive rule pr, of an agent a, set the state condition, the auxiliary condition, and the set of actions of the rule. The set of actions will include the sending of the message that have been considered to define the rule.
- For each agent a define the auxiliary predicates. If the agent is an interface agent, one auxiliary predicate must define the agent interpreter.

3.4.7 Step 7: prototype implementation

This step defines the (almost) executable ProlAg code for each agent specified in the previous steps. In particular, the interfaces to external software and data must be defined here in terms of interpreters.

3.4.8 Step 8: prototype execution, testing and simulation

This step is performed using the CaseLP Visualizer or the CaseLP Shell. The prototype is executed, possibly many times, in order to *validate* its behaviour against the client's desiderata. Execution provides the client and the developer with information about all the events that occur within the MAS, allowing them to check that the prototype behaves as expected and that it contains all the desired features. Any error or misbehavior discovered in this step may imply a revision of some of the choices made in the previous steps and a refinement of the application against the initial requirements.

3.5 Applications of CaseLP

This section outlines two prototypes that have been realized using CaseLP. These case-studies concern different application domains and have been realized in co-operation with Elsag S.p.A., an Italian company that provides automation services internationally. A complete description of the development of a prototype in CaseLP is presented in Chapter 6, where the case-study concerns the management of freight train traffic along a European railway line. That application was realized in co-operation with Italian Railways (FS S.p.A).

3.5.1 Planning of goods transportation

The prototype presented in [55] models a multi-agent scenario in which the transportation of goods from an European city to another must be optimized. In the scenario there are the following entities:

- a *client*, that requires the transportation of good, optimized in terms of delivery time or cost;
- an *agency*, which receives the request from the client, forwards it to one of the distributors present in the departure city, and informs the client as to whether or not the delivery is possible.
- a set of *distributors*, which are located in a city and which manage transportation within a part of the European territory.
- a set of heterogeneous *transporters*, which operate on tracks between pairs of cities and perform the physical transportation of goods on behalf of the distributors.

CaseLP was used in this context to realize a prototype that modelled the concurrent submission of requests for transportation services from several clients. The prototype was profitably used to monitor the messages exchanged among the above entities (modelled as agents) in order to gather information concerning different transportation possibilities and to reserve the resources (transporters) needed to perform them.

3.5.2 Vehicles monitoring

The prototype presented in [3] models an application for the remote surveillance and control of traffic within service areas on an Italian motorway. The application employs automatic car detection and number-plate reading at monitored gates.

The scenario for vehicle monitoring involves a large number of heterogeneous entities. These entities can be human (the policeman who inserts a new number-plate to be monitored or who receives an alarm concerning a suspect vehicle), physical (remote cameras for getting number-plates images, communication infrastructure) and virtual (the client interface, the application server and the remote sensor control with image processing for license plate reading). The configuration is distributed on a protected TCP/IP WAN, using cables and satellite links, each with limited bandwidth. Due to this restriction, data allocation and communication strategies had to be modeled in advance, in order to optimize the system before its physical implementation on the real WAN. Moreover, the modelling of each entity's behavior was needed to pre-test the entire message exchange system in terms of coherence, deadlock absence and performance.

Application prototyping by means of CaseLP proved to be very effective in achieving these objectives. The prototype was used to improve the application specifications, to properly dimension the communication channels, and to provide a formal specification for use during the implementation stage. The implementation was then completed successfully and the vehicle monitoring apparatus was installed on the motorway section between Salerno and Reggio-Calabria (Italy). It is currently used by the Italian Police, Financial Police and by the "Autostrade" (motorway) organization for traffic monitoring.

Chapter 4

Ontologies in CaseLP

This chapter discusses an approach for adding explicit ontologies to multi-agent systems based on logic programming. This approach has been described in [88]. Ontologies are content theories about knowledge domains, developed to clarify knowledge structure and enhance knowledge reuse and standardization. Ontologies allow for the explicit organization of knowledge in agent-based applications, as well as for the unambiguous description of characteristics and properties of agents.

We consider in detail the use of explicit ontologies in CaseLP. Our running example comes from the domain of sport results. Concepts such as sport, competition, competitors are included in the domain ontology, as well as the relationships that relate these concepts. We also introduce an agent level ontology to formalize attributes and functionalities of CaseLP agents, including their kind, architecture and services, both at the domain level and at the agent level. Domain and agent level ontologies can be exploited in CaseLP to perform semantic checks on agent architectural descriptions, to check agent behavioural rules used by an agent to provide its services, and as a knowledge repository to be exploited during agent execution.

4.1 Ontologies and their use in multi-agent systems

Ontologies are *content* theories about objects, their properties, and the relationships between them that are possible in a specific domain of knowledge [13]. In knowledge-based systems, an ontology is that part of the system which specifies what things exist and what is true about them. Ontological analysis clarifies the structure of knowledge. An ontology related to a domain forms the heart of any system of knowledge representation for that domain. Without ontologies, there cannot be a vocabulary for representing knowledge.

Ontologies enable knowledge sharing, since they capture the intrinsic conceptual structure of the domain using a suitable knowledge representation language. This language can be shared with others that have similar needs for knowledge representation in that domain, thereby eliminating the need for replicating the knowledge-analysis process.

In the setting of knowledge-based agents and multi-agent systems, an ontology is a description of the concepts and relationships that can exist for an agent or a community of agents. Roughly speaking, ontologies specify the vocabulary used to talk about both the agents, their characteristics and functionalities, and the domain in which they operate. An ontology is an "official" set of attributes and relationships related to the agents and their domain of application.

The way knowledge is embedded in agents is the essence of *knowledge organization*. Explicit ontologies make the knowledge-based organization in a MAS clear. An ontology is designed for the purpose of enabling knowledge sharing and reuse. An ontological commitment is an agreement among agents to use a vocabulary in a way that is consistent (but not complete) with respect to the theory specified by an ontology.

From the point of view of a single agent in a MAS, three different sources of knowledge may be necessary:

- domain knowledge;
- agent knowledge;
- computational knowledge.

Agents manipulate information related to some particular application domain. Domain knowledge can be incorporated into a domain ontology. A proper structuring of domain dependent knowledge could be profitably shared among different agents and/or reused for building multi-agent systems that operate on similar domains, either from the point of view of enclosed objects, or the point of view of structuring of relationships among objects.

On the other hand, agents need to know characteristics and properties of other agents with which they interact. Agent knowledge is expressed in an agent ontology, which describes features that do not depend on the domain to which a multi-agent system is applied. An agent ontology can be used to realize several MAS that operate on different application domains. Furthermore, as anticipated in Section 2.1, the most debated issue in the agent community is still a clear definition of what agents and multi-agent systems are [30]. A suitable agent ontology could certainly help in the definition of a set of general and unambiguous features to characterize different classes of agents.

In this chapter we model the knowledge that may be included in a domain ontology for an application that finds the "best" sporting city in Italy. Moreover, we present an agent level ontology for agents that can be defined in CaseLP. Domain and agent level ontologies are joined in CaseLP to allow analysis of MAS architectural descriptions and single agent behaviours. Agents exploit domain level ontologies during computation as knowledge repositories.

This chapter is structured as follows. The next section presents the ontology we have defined for the domain of sport results. Section 4.3 introduces the ontology we have defined for CaseLP agents. Section 4.4 gives some examples on how ontologies can be used in CaseLP. Finally, Section 4.5 briefly discusses the benefits and drawbacks of our work and compares it to other proposed approaches for the building of ontologies.

4.2 A domain ontology for sport results

Multi-agent systems have a lot of potential for improving our interaction with the Internet. For example, in [78], the activities of the Intelligent Agent Lab at the University of Melbourne on information finding agents are described. One of these activities involves applying agents to a prototypical information finding task (the finding of sport scores) on the Internet. The domain of sports results is interesting as it highlights the contrast between the uniformity and diversity of information available on the Web.

The starting point for a MAS prototype capable of finding the "best" sporting city in Italy (by searching and analyzing a number of sport information sources), could be *SportsFinder* [47], an information agent which extracts sports results from the Web. The overall prototype could behave as follows: the score for a city would be calculated using a heuristic function; partial scores would be calculated by $aggregator$ agents, one for each sport, that would interact with a $result\_finder$ agent to collect results related to their particular sport; a $best\_finder$ agent would receive partial scores from each aggregator agent and calculate the final score for each city; and, the names of cities achieving a maximum score would then be returned to the user, along with their score.

4.2.1 Modelling the domain ontology

Relevant objects for this application domain are sports, competitors, competitions and cities. They are incorporated in the domain ontology we have defined. The natural way of modelling an ontology for an agent-based system based on logic programming is as a 5-tuple

$$< name, terms, predicates, kb, constraints >$$

where *name* identifies the ontology, *terms* is a set of constants and function symbols, *predicates* is a set of predicate symbols to be applied to terms, the knowledge base *kb* is a set of definite Horn clauses, and finally *constraints* is a set of constraints that *kb* has to satisfy.

For our running example, the name of the ontology is *sports*. The terms are a union of the constants denoting sports, competitors, competitions, competition classes, and competition types. There are no function symbols in *sports*. The predicates are *sport*/1, *competitor*/1, *type*/1, *class*/1, *competition_of*/2, *competition_type*/2, *competition_class*/2, *participant*/2, and *plays*/2.

Part of the knowledge base of the ontology *sports* is presented below. The knowledge base contains facts for every sport. They are

$$\begin{array}{l} sport(cycling), \\ sport(soccer), \\ sport(basketball), \\ sport(volleyball). \end{array}$$

Each sport is associated to a set of competitions that, in turn, are associated to a competition type and a competition class. Competition type distinguish competitions organized as a series of matches between pair of competitors (soccer, basketball, volleyball) from competitions whose result is a ladder containing all the competitors (cycling). Competition class refers to the importance of a competition. For example, "Serie A" is more important than "Serie B" and "Tour de France" is more important than whatever cycling world cup race.

Part of the corresponding section of knowledge base is

$$\begin{array}{l} competition\_of(serie\_a, soccer), \\ competition\_of(serie\_b, soccer), \\ competition\_type(serie\_a, match), \\ competition\_type(tour\_de\_france, ladder), \\ competition\_class(serie\_a, 1), \\ competition\_class(serie\_b, 2). \end{array}$$

The arguments of the predicate *competition_of* are a competition and a sport, while the arguments of *competition_type* and *competition_class* are a competition and a type, and a competition and a class, respectively.

Each competition has participants. We define a predicate *participant* that holds for a competitor and a competition. For example,

$$\begin{array}{l} participant(marco\_pantani, giro\_italia), \\ participant(torino, serie\_a) \end{array}$$

are part of the knowledge base. Participants of a competition are competitors, so the predicate *competitor* is defined as

$$competititor(X) \leftarrow participant(X, Y) \wedge competition\_of(Y, Z).$$

Furthermore, each competitor plays a sport, then

$$plays(X, Y) \leftarrow participant(X, Z) \wedge competition\_of(Z, Y)$$

is included in the knowledge base.

We also need an ontology for cities. This ontology consists of the names of the cities and the names of the competitors (as constants), and two predicate symbols $city/1$ and $is\_from/2$ which hold for terms representing cities and for competitor-city pairs respectively. The knowledge base for the city ontology is expressed by facts such as

$$\begin{array}{l} city(torino), \\ city(bologna), \\ is\_from(torino, torino), \\ is\_from(marco\_pantani, bologna). \end{array}$$

The ontology for the multi-agent system is the union of the sports and city ontologies. Note that the participant names are assumed to be the same.

The set of constraints is the union of the constraints belonging to both the sport ontology and the city ontology. Furthermore, new constraints can be defined. For example,

$$\begin{array}{l} plays(X, Y) \rightarrow competitor(X) \wedge sport(Y), \\ is\_from(X, Y) \rightarrow competitor(X) \wedge city(Y) \end{array}$$

are two constraints. The first constraint states that relationships *plays* holds on a competitor and a sport, the second one that is_from holds between a competitor and a city. Constraints can be exploited to check knowledge base consistency whenever the ontology is updated.

4.3 Adding ontologies to CaseLP

4.3.1 Ontology *agent*: domain independent part

As already stated in Section 3.2, a CaseLP agent is characterized by a *kind*, an *architecture*, a possible *interpreter*, and set of *services*. The agent level ontology contains domain independent knowledge. Values for attributes defining *kind, architecture, and interpreter* do not depend on the particular domain of application for which the prototype is being constructed. Instead, they depend solely on the fact that agents are CaseLP agents. Values for *kind* define the type of an agent. This value constrains the possible further attributes that the agent has and the values that attributes can assume. For example, the attribute *interpreter* is defined only for interface agents and interface agents have necessarily a reactive architecture.

We will now define an agent level ontology, that we simply call *agent*. First, we need terms for expressing the kinds of agents available and a predicate that says these terms are proper values for agent kinds. The natural choice here is to define four constants

$$logical, interface, facilitator, manager$$

and a predicate

$$agent\_kind/1.$$

The knowledge base of ontology *agent* will then contain four facts of the form

$$\begin{array}{l} agent\_kind(logical), \\ agent\_kind(interface), \\ agent\_kind(facilitator), \\ agent\_kind(manager). \end{array}$$

Similarly, we need to express the available agent architectures. We have the terms

$$reactive, proactive, reactive\_proactive$$

and the predicate

$$agent\_architecture/1.$$

The corresponding part of knowledge base is

$$\begin{array}{l} agent\_architecture(reactive), \\ agent\_architecture(proactive), \\ agent\_architecture(hybrid(reactive\_proactive)). \end{array}$$

Each of the above terms refers to a a standard CaseLP architecture. New terms will be enclosed into the ontology as soon as other architectures are available in CaseLP.

Each interface agent is given one or more interpreters for use in accessing external legacy modules. Predicate *interpreter*/1 holds for terms that represent possible interpreters.

In addition to the previously mentioned features, each agent provides a set of basic services. These services are provided implicitly by certain CaseLP agents, but have to be made explicit by the agents that require them. Agent basic services can include:

- creation or deletion of an agent;
- provision of information about agent kind, architecture, services;

- link of a service provided by an agent with a service required by another agent;
- search for an agent that provides some services.

The ontology *agent* includes vocabulary to identify basic services. Predicate

$$agent\_service/1$$

holds for all the terms that represent basic services provided by agents, and the corresponding part of the knowledge base contains predicates such as

$$\begin{array}{l} agent\_service(createag), \\ agent\_service(destroyag), \\ agent\_service(agent\_type), \\ agent\_service(architecture\_type), \\ \ldots \end{array}$$

As described in Section 3.3, for each service a specific conversation needs to be defined between the agent that provides the service and the one that requires it. A conversation is composed of KQML messages which are exchanged between pairs of agents. Terms used as *contents* of KQML messages and the predicates $agent\_msg/1$ and $associated\_to/2$ are defined in the *agent* ontology. These predicates identify terms as message contents and associate message contents to services respectively. Advantage is taken of variables in logic programming to allow for incomplete message contents where answers can be returned. For example, the term for expressing the message content related to the service *agent_type* is

$$agent\_type(Type).$$

This term is the content of a KQML message that may be sent by an agent a to another agent b to ask b's kind. The variable $Type$ will be instantiated by b before replying to a. The contents of the knowledge base related to $agent\_type(Type)$ are

$$\begin{array}{l} agent\_msg(agent\_type(Type)), \\ associated\_to(agent\_type(Type), agent\_type). \end{array}$$

To give another example, the service *creatag* is associated to the message content

$$createag(Agentname, Kind, Arch, Int, Provserv, Reqserv, Expserv, Impserv)$$

and the service *destroyag* to the message content

$$destroyag(Agentname).$$

These messages commit the *manager* agent that receives them to create a new agent called *Agentname* with characteristics provided, and to delete the agent named *Agentname* respectively. Finally, the service *who_provides_service* is associated to the message content

$$provide(Service, Agentname).$$

This request is received by a *facilitator* agent, which instantiates the variable *Agentname* before replying to the agent that submitted the request.

Not all kinds of agents provide the same set of services. Thus, relationships to relate services to particular kinds of agents are needed in the ontology. For example the fact,

$$service\_of(createag, manager)$$

states that creation of a new agent is a service provided by manager agents. Similarly, the fact

$$service\_of(who\_provides\_service, facilitator)$$

states that the service *who_provides_service* is offered by facilitator agents.

The predicate $architecture\_of/2$ defines a relationship between different kinds of agents and their allowable internal architectures. For example, it would not make sense for an interface agent to have a sophisticated architecture. This is because the task of interface agents is merely to translate requests from other agents in the MAS into queries that can be submitted to the external modules to which they are linked. A simple reactive architecture suits this task. On the other hand, logical agents can be built using all available internal architectures. Their task is coordinating and controlling activities of the MAS. According to the complexity of this task, the suitable architecture can be chosen. Similar considerations can be made for facilitator and manager agents. Facilitators reactively respond to questions about service providers so they can be thought of as purely reactive agents. Managers can either simply react to creation request by performing initialization of new agents or they can act autonomously by creating new agent instances, according to their own internal states and plans. Their architecture should therefore be as flexible as possible.

The remarks above can be captured in the ontology knowledge base by the following facts.

$$\begin{array}{l}
architecture_of(logical, reactive),\\
architecture_of(logical, proactive),\\
architecture_of(logical, reactive_proactive)),\\
architecture_of(interface, reactive),\\
architecture_of(facilitator, reactive),\\
architecture_of(manager, reactive),\\
architecture_of(manager, proactive),\\
architecture_of(manager, reactive_proactive)).
\end{array}$$

4.3.2 Ontology *agent*: domain dependent part

So far, we have presented the domain independent part of the ontology for CaseLP agents. Besides the agent level services already presented, agents have at their disposal domain level services that are related to the application domain of the MAS. In the following, we define domain level services and message contents for the domain of sports results, as well as constraints on the ontology. We introduce the predicate

$$domain\_service/1$$

to express that some terms represent services at the domain level. Predicate $domain\_service$ represents a "link" between the agent level and the domain on which an agent operates. Furthermore, we need a predicate

$$domain\_msg/1$$

to express that some terms denote domain level message contents, as well as a predicate

$$domain\_associated\_to/2,$$

to associate domain level message contents to domain level services.

Part of the ontology knowledge base for domain services for the sports application would be

$$\begin{array}{l} domain\_service(last\_result), \\ domain\_service(last\_ladder\_position), \\ domain\_service(city\_sport\_score), \\ domain\_service(best\_cities), \\ domain\_service(best\_city\_score). \end{array}$$

The service $last\_result$ is provided by $result\_finder$ agents and is used by $aggregator$ agents to retrieve the last result obtained by a given competitor in a given competition. The service $last\_ladder\_position$ is also provided by $result\_finder$ agents and is used by $aggregator$ agents to retrieve the last position achieved by a given competitor in a given competition. Aggregators combine these two results to calculate the competitor's score in the given competition. Competitors' scores are in turn combined to obtain the city score for a particular sport. The city score for a particular sport is transferred from $aggregator$ agents to the $best\_finder$ agent via the service $city\_sport\_score$. The services $best\_cities$ and $best\_city\_score$ are exported by the $best\_finder$ agent to provide information to external users.

For each of the above (not exported) services, we include some terms for message contents in the *sports* ontology. For example, the content message for the $last\_result$ service and its association to the service are expressed as

$$domain\_msg(last\_result(Sport, Competition, Competitor, Result)),$$
$$domain\_associated\_to(last\_result(Sport, Competition, Competitor, Result), last\_result)$$

Similarly, the message content for the *city_sport_score* service and its association to the service are expressed as

$$domain\_msg(score(City, Sport, Score)),$$
$$domain\_associated\_to(score(City, Sport, Score), city\_sport\_score).$$

The last component of an ontology is a set of constraints on the knowledge base. For example, the relationship *associated_to* holds only between an *agent_msg* and a *service*. This is captured by the constraint

$$associated\_to(X, Y) \rightarrow agent\_msg(X) \wedge service(Y).$$

Analogously, the relationship *service_of* holds only between a *service* and a *kind*. The corresponding constraint is

$$service\_of(X, Y) \rightarrow service(X) \wedge kind(Y).$$

Similar constraints can be set to state other properties that the knowledge base has to satisfy.

As far as domain level is concerned, the constraint

$$domain\_associated\_to(X, Y) \rightarrow domain\_msg(X) \wedge domain\_service(Y).$$

states that relationship $domain\_associated\_to(X, Y)$ holds on domain message contents and domain services.

In the next section, we give some ideas about how the ontologies we have defined can be used in CaseLP.

4.4 Exploiting ontologies in LP agents

The ontology *agent* can be exploited to perform a semantic check on an architectural description of a CaseLP MAS[1]. In CaseLP, the domain-independent architectural description for an agent class is

[1]See Section 3.3 for more details on CaseLP MAS architectural description

```
agentclass <ClassName> {
    kind: <Kind>;
    architecture: <ArchType>;
    interpreter: <IntName>;
}
```

In order to realize a MAS prototype, various classes of agents are defined. From class descriptions, the process of deriving a theory T_{ad} is straightforward. The theory T_{ad} is a clausal representation of the architectural description of a MAS. For each class definition, it contains clauses of the form

$$\begin{aligned}&agent\_class(ClassName) \leftarrow\\&\quad agent\_kind(Kind) \wedge\\&\quad agent\_architecture(ArchType) \wedge\\&\quad architecture\_of(Kind, ArchType) \wedge\\&\quad interpreter(IntName).\end{aligned}$$

We can check semantic consistency of this theory proving that

$$agent_{kb} \models T_{ad}$$

where $agent_{kb}$ is the knowledge base of the *agent* ontology. For example, if we define class $result\_finder$ in which `kind` assumes the value *logical*, this definition does not pass the semantic check, because logical agents have not an interpreter associated with them. The same happens if we define a $facilitator$ agent with a *proactive* architecture, or if the service *createag* is linked to a non-manager agent.

We can use the domain dependent part of the ontology *agent* to check the rules that define the program of an agent. Every reactive *event-condition-action* rule that is defined in the agent program, has to be related to a service provided by the agent. For example, the event part of the rule used by a $result\_finder$ agent to provide the service *last_result* is the following:

$$\begin{aligned}&on\ message\\&\quad ask(content(last\_result(Sport, Competition, Competitor, Result)),\\&\qquad sender(Aggr))\end{aligned}$$

The content of the message in the head of the rule has to be a message content that is associated to the mentioned service.

As far as domain-dependent ontologies are concerned, predicates that are defined in the domain ontology can be exploited in an agent's behaviour as *conditions* that must be satisfied in order to proceed with computation. For example *aggregator* agents use the predicates *plays* and *participant* from the *sports* ontology to retrieve all the players of the sport under their responsibility, and to find out in which competitions these players participate, respectively. The agent $best\_finder$ uses the predicate is_from to select all players that live in a particular city.

4.5 Discussion

be defined in the setting of CaseLP, a framework based on logic programming. Logic programming has undoubtedly many characteristics, which make it a suitable paradigm for implementing ontologies. One example is that LP languages naturally provide for the declarative representation and organization of symbolic knowledge. LP incorporates inference and reasoning capabilities that can be exploited either to derive information based on ontological knowledge, or to verify whether or not ontology constraints are satisfied. In CaseLP agents are implemented as logical modules. Ontologies are implemented as logical modules as well, and thus can be easily included into agent code.

At the moment, ontologies are used in the CaseLP prototyping method to model the knowledge related to the prototype application domain. A full integration into the CaseLP framework of tools for implementing the various kinds of analysis outlined in Section 4.4 is under study. At present, semantic check of a MAS architectural description can be executed after it has been defined according to the CaseLP prototyping method. A check on the program rules of the agents is performed using the domain dependent part of the *agent* ontology. This check can be executed during the translation of the AgentRules specifications into ProlAg, properly exploiting the agent level ontology. This ontology reflects the current definition of CaseLP agents. Even though we believe that this definition is extensive enough, it could be changed in the future to include new useful agent features. In this case, the ontology would be properly updated.

Our approach to the definition of ontologies is certainly bottom-up. For the definition of the ontology *agent* our starting point was the particular model represented by CaseLP agents and to define the ontology *sports* it was the specific domain of the application. This is certainly a drawback, since we had to start from scratch to create an organization of concepts which could have been borrowed from more *general* ontologies. Research on *general* ontologies (see [13] for a survey) has led to *task-independent* ontologies, for example CYC, Wordnet, Um-Thing and Sowa's. Currently an ontology cannot cover all possible potential uses, and choosing the most appropriate general ontology for a particular application domain is problematic at best. General ontologies are huge tree structures of concepts. If we cannot identify a suitable subtree for our application domain, we are obliged to include a very large set of concepts that may not be of interest.

The decision to keep agent and domain ontologies separate was driven by the necessity to facilitate ontology management and improve ontology comprehension and standardization.

Features captured by the ontology *agent* enhance the comprehension of what a CaseLP agent is. Moreover, the ontology *agent* can be used a starting point for the definition of a more general ontology that represents and organizes characteristics of a broader class of agents and MAS. As far as domain ontologies are concerned, this separation keeps them as independent as possible from their use in a particular MAS. Knowledge in a domain ontology can be defined without regard to how it will be manipulated by CaseLP agents, which certainly enhances reuse and standardization. The agent and domain levels have to be linked, and part of the agent ontology must include terms that will be manipulated at domain level. The predicates *domain_service*, *domain_msg* and *domain_associated_to* have been included in the agent ontology for this purpose.

Chapter 5

Languages for high-level specification of agent architectures

As advocated by the ARPEGGIO project (see Section 2.4), one of the initial ideas in the design of CaseLP was to create an environment which could accommodate for various specification and implementation languages as well as for legacy software, so that a MAS prototype could be built using a range of techniques integrated into a common framework based on logic programming.

The flexibility for a software engineer to be able to choose the most suitable language for modelling different aspects of a MAS can improve the overall realization of agent-based applications. In particular, in the realization of a MAS the developer may need to use heterogeneous agent architectures, so that each application component can be individually optimized for performing its task. Languages that can easily model heterogeneity in agent architectures are therefore very useful in general.

CaseLP provides the prototype developer with standard reactive, proactive, and reactive-proactive architectures. If the prototype under development would need to incorporate a non-standard architecture, this new architecture should first be specified, then verified and/or tested, and finally implemented.

In this chapter we investigate the use of two high level specification languages to describe sophisticated agent architectures. These two languages have still to be fully integrated into CaseLP but represent good candidates to be used in the CaseLP method when the specification of specialized agent architectures is needed. We propose a micro-level development method for the specification of the internal structure and functioning of agents, to be taken into consideration in the development of a MAS only if specialized agent architectures are required for the realization of the prototype.

5.1 Micro-level prototyping method

The architecture chosen for each agent template in Step 4 of the CaseLP method may refer to a non standard architecture that still has to be defined. If this is the case, then the following method could be used for the specification, verification, testing, and implementation of a new agent architecture[1].

1. The first stage uses HEMASL or $\mathcal{E}_{hhf}$ to specify the internal structure and functioning of the new architecture. The specification defines the internal components containing the state and the program of the agent. Moreover, the specification defines the architecture engine.

2. The specification of the architecture can be formally verified or validated via animation. If the specification is in HEMASL, it is first translated into $\mathcal{E}_{hhf}$. Since $\mathcal{E}_{hhf}$ is a logical *executable* language, both the verification and validation of architecture properties are possible.

3. After being verified and validated, the architecture specification is translated into ProlAg, in order to be used in the prototype.

5.2 The linear logic language $\mathcal{E}_{hhf}$

The linear logic language $\mathcal{E}_{hhf}$ [23, 24] can be used to specify agent architectures, including their data structures, and rules that model their execution engines. Specifications in $\mathcal{E}_{hhf}$ are executable, so it is possible to directly interpret a given specification. In this way it is possible to provide a better understanding of the system, and to facilitate the refinement of the specification to a point where it behaves exactly as expected. Moreover, the language $\mathcal{E}_{hhf}$ can be used to properly characterize an agent architecture from a semantic point of view, since linear logics naturally models concurrent and state-based systems, which in the end, is exactly what agents are.

Here, we briefly present the main features of $\mathcal{E}_{hhf}$. Section 5.1 presents an example of architecture specification in $\mathcal{E}_{hhf}$, along with an approach for translating high-level $\mathcal{E}_{hhf}$ specifications into the low-level logical language ProlAg, which is used in CaseLP for the implementation of prototypes[2].

The language $\mathcal{E}_{hhf}$ is an executable specification language for modeling concurrent and resource sensitive systems, based on the general purpose logical specification language Forum

[1]This micro-level method has not been experimented in depth in CaseLP. In particular, the translation between HEMASL and $\mathcal{E}_{hhf}$ has to be investigated in depth.

[2]See Section 3.3.6 for a description of ProlAg.

[61]. $\mathcal{E}_{hhf}$ is a multiset-based logic language, which combines features of logic programming extensions like λProlog (e.g. goals with implication and universal quantification), with the notion of *formulas as resources* that is the basis of linear logic [33]. A $\mathcal{E}_{hhf}$-program P is a collection of multi-conclusion clauses of the form:

$$A_1 \parr \ldots \parr A_n \multimapinv Goal,$$

where A_i is an atomic formula, and the linear disjunction $A_1 \parr \ldots \parr A_n$ corresponds to the head of the clause, with $Goal$ as its body. Furthermore, $A \multimapinv B$ is a linear implication. Clauses of this kind *consume* the resources (formulas) they need in order to be applied in a resolution step.

Given a multiset of atomic formulas (the state of the computation) Ω_0, a resolution step $\Omega_0 \rightarrow \Omega_1$ can be performed by applying an instance $A_1 \parr \ldots \parr A_n \multimapinv G$ of a clause in the program P, whenever the multiset Θ consisting of the atoms $A_1, \ldots, A_n$ is contained in Ω_0. Ω_1 is then obtained by removing Θ from Ω_0 and by adding G to the resulting multiset. In the $\mathcal{E}_{hhf}$ interpreter, instantiation is replaced by unification. At this point, since G may be a complex formula, the search rules (i.e., the logical rules of the connectives occurring in G) must be exhaustively applied in order to proceed. Such a derivation corresponds to a branch of the proof tree of the multiset Ω. Ω represents the current global state, while P describes a set of clauses that can be triggered at any point during a computation.

$\mathcal{E}_{hhf}$ provides a way to "guard" the application of a given clause. In the extended type of clauses

$$G_1 \,\&\ldots\&\, G_m \;\Rightarrow (A_1 \parr \ldots \parr A_n \multimapinv Goal),$$

the goal-formulas G_i are *conditions* that must be solved in order for the clause to be triggered.

New components can be added to the current state by using goal-formulas of the form $G_1 \parr G_2$. The goal $G_1 \parr G_2, \Delta$ simply reduces to G_1, G_2, Δ. Conditions over the current state can be tested by using goal-formulas of the form $G_1 \& G_2$. The goal $G_1 \& G_2, \Delta$ reduces to G_1, Δ and G_2, Δ. Thus, one of the two copies of the state can be consumed to verify a contextual condition. Universal quantification in a goal-formula $\forall x.G$ can be used to create a new identifier t which must be local to the derivation tree of the subgoal $G[t/x]$. Finally, the constant **ALL** succeeds in any context and the constant $\perp$ is simply removed from the current goal.

Architecture specification in $\mathcal{E}_{hhf}$ can be tested using the $\mathcal{E}_{hhf}$ interpreter. Testing is performed in order to verify to what extent the architecture prototype corresponds to the desired requirements. This may lead to changes, improvements or refinements being made to the design of the architecture. Using a logical language like $\mathcal{E}_{hhf}$ as the specification language provides several advantages such as:

- It is possible to evaluate a goal in a step by step manner, following the evolution of a particular system (architecture) in detail.
- Through *backtracking* it is possible to follow all the different evolutions of a given system (architecture), depending for example on the order of application of clauses that specify the engine rules. It is therefore possible to simulate a multi-threading architecture, where various components execute concurrently.
- It is possible to verify whether a particular computation may be carried out, or, more importantly, that *every* computation starting from a given configuration leads to a final state satisfying a given property, independently of the order of execution of architecture components. To this aim it suffices to run the desired computation together with a goal negating the desired property of the final state, and then to check whether the global goal fails.

In [4] a $\mathcal{E}_{hhf}$-based framework for MAS is defined. It is used to model MAS in which agents with heterogeneous architectures can co-exist.

5.2.1 Specification and implementation of the BDI engine

As an example of the use of $\mathcal{E}_{hhf}$ for architecture specification, we present the specification of an engine for a BDI architecture [7]. The specification is based on [25], a specification of the BDI architecture in **Z**. An approach for the translation of $\mathcal{E}_{hhf}$ rules into ProlAg code is also presented.

5.2.1.1 $\mathcal{E}_{hhf}$ specification of a BDI architecture

We now explain in details how to specify the salient aspects of the BDI architecture using $\mathcal{E}_{hhf}$.

Beliefs, goals, actions, and plans. *Beliefs* can be modelled by ground facts of the form *belief(Fact)*. The set of beliefs is maintained in the current state Ω. *Goals* are terms of the form *achieve(Fact)* and *query(Fact)*. An *achieve goal* commits the agent to a sequence of actions, whose execution must make the goal true. A *query goal* implies a test on the agent's beliefs to know whether the goal is true or not. *Internal actions*, represented by the terms *assert(Fact)* and *retract(Fact)*, update the agent beliefs. *External actions* (e.g. sending messages) are denoted by generic terms. *Plans* basically consist of sequences of actions and (sub)goals. They are represented as facts in the following form

$$plan(Trigger, Context, Body, Maintenance, SuccActs),$$

where *Trigger* is a trigger linked to a particular event, *Context* is a condition that has to be true to start the plan, *Body* is a sequence of actions and (sub)goals $B_1 :: \ldots :: B_n$, *Maintenance* is a condition that has to be true while the plan is executed, *SuccActs* is a sequence of actions to be executed if the plan succeeds. Thus, the *plan library* of the agent is described as a collection of facts. Differently from [25] we do not take into consideration *failure actions* whose aim is generally to rollback to a safe state in the case of plan failure. This can be obtained without effort in an LP setting by exploiting backtracking.

Triggers and events. Triggers are raised by events and their aim is to activate plans. We can distinguish between *external triggers*, that force an agent to adopt a new intention, and *internal triggers*, that cause an agent to add a new plan to an already existing intention. External triggers are: *addbelief(Fact)* and *delbelief(Fact)*, linked respectively to events denoting acquisition and removal of a belief, and *goal(G)* denoting acquisition of a new goal. There is only one internal trigger, i.e., *subgoal(G)*, which denotes a (sub)goal call in the executing plan. Events are terms of the form *event(Trig, Id)*, where *Trig* can assume one of the forms above, and *Id* is a (possibly undefined) identifier of the plan instance causing the event.

Event queues. An event queue is associated with each BDI agent. It contains external and internal events and is represented by a term of the form

$$event\_queue([E_1 \mid \ldots \mid E_n]),$$

where $E_1 \ldots E_n$ are events. Events linked to external triggers are always inserted at the end of the event queue, whereas events linked to internal triggers are inserted at the top. Since events are taken from the top of the event queue, this policy gives priority to events generated by an attempt to achieve a (sub)goal.

Plan instances. Plan instances represent plans to be executed. A new plan instance is created and inserted into an *intention stack* as soon as its trigger has been activated and its context is satisfiable. A plan instance contains an instantiated copy of the original plan, from which it derives, along with information about whether the plan is *active* or *suspended*. Differently from [25], we use shared variables and unification to inherit knowledge from the original plan. A unique identifier is associated to each plan instance. Thus, plan instances have the form

$$plan\_inst(Id, Body, Maintenance, Active, SuccActs).$$

Intention stacks. An intention stack contains the currently active plan and a set of suspended plans. An internal trigger *subgoal(G)* can generate a new plan instance that is pushed onto the same intention stack where as the plan containing the (sub)goal call which activated the trigger. An external trigger produces a new plan instance that is stored in a new intention stack. In our representation an intention stack is a term of the form

$$int\_stack([P_1 \mid \ldots \mid P_n]),$$

where $P_1 \ldots P_n$ are plan instances. The internal behaviour of a BDI agent can be described by means of the $\mathcal{E}_{hhf}$ rules listed below. For the sake of this example we do not explicitly give rules for *perception*, i.e., rules for simulating interactions with the external world. As in [25], the idea is to start the simulation of the agent from an initial queue of events and from an initial intention stack. During execution, perception rules can be used to build appropriate events according to information produced by some perception devices (for example environment sensors or inter-agent communication devices) and to (non deterministically) post them into the agent event queue. In addition to the rules listed below, we consider a program P that contains the plan library and the information needed to distinguish between external and internal triggers and actions, of the form $external(Trigger)$, $external(Action)$, etc.

Plan triggering. Plan triggering rules handle the creation of new plan instances and new intention stacks, according to the type of raised trigger. When an external trigger is handled, a new intention stack is created and a new plan instance is pushed onto it. This process can be formalized by the following rule:

$$\begin{array}{l} plan(T, C, B, M, SA) \;\&\; external(T) \;\Rightarrow\; (event\_queue([event(T, \_)|L]) \;\circ\!- \\ \quad verify(C) \;\&\; (\forall Id.\; int\_stack([plan\_inst(Id, B, M, act, SA)]) \;\invamp\; event\_queue(L))). \end{array}$$

The conditions $plan(\ldots)$ and $external(\ldots)$ must be fulfilled by P (i.e., must be unified with facts in P), whereas the goal-formula $verify(\ldots) \;\&\; (\forall Id. \ldots)$ allows us to test the contextual condition over a copy of the current state (for the sake of brevity $verify$ is not specified here). New identifiers for the plan instances are created by using universal quantification. Finally, note that the modification of the event queue is defined by *consuming* the current one (the head of the clause) and creating a new copy (in the body).

A similar clause formalizes what to do if the trigger is internal. The main difference is that the intention stack, whose top plan has the same identifier as the first event in the event queue, is *consumed*. After verifying the context, the intention stack is rewritten by adding a new instance of the plan whose trigger matches the trigger of the topmost event in the event queue.

Plan execution. A plan is executed when the event queue of the agent is empty. An intention stack such which has an active top plan instance is chosen non deterministically, and the first action of the plan instance is executed (if the maintenance condition is verified). We have developed rules for each possible plan component: external and internal actions, query goals and achieve goals. We report the most significant among them.

In the case where an external action has to be executed, the following rule is applied:

$$\begin{array}{l} external(A) \;\Rightarrow\; (event\_queue([]) \parr int\_stack([plan\_inst(Id, A :: B, M, act, SA)|L]) \\ \quad \circ\!- \;\; verify(M) \;\&\; (event\_queue([]) \parr execute(A) \parr \\ \qquad int\_stack([plan\_inst(Id, B, M, act, SA)|L]))). \end{array}$$

The call to $execute(A)$ activates the agent output devices, for example effectors on the environment, or the inter-agent communication system used for sending a message to another agent.

In the case of internal actions the following rules are applied; if the top component of a plan is an *assert* action, the corresponding belief is added to the database (if it is not already present):

$$\begin{array}{l} event\_queue([]) \parr int\_stack([plan\_inst(Id, assert(F) :: B, M, act, SA)|L]) \;\circ\!- \\ \quad verify(M) \;\&\; not\_believed(F) \;\&\; (belief(F) \parr \\ \quad event\_queue([event(addbelief(F), Id)]) \parr \\ \quad int\_stack([plan\_inst(Id, B, M, active, SA)|L])). \end{array}$$

Similarly, if the top component of a plan is a *retract* action, then the corresponding belief (if present) is removed form the database by *consuming* it.

If the plan component is an *achieve* goal, and the corresponding belief is in the database, then the agent can proceed:

$$\begin{array}{l} event\_queue([]) \parr belief(F) \parr \\ \quad int\_stack([plan\_inst(Id, achieve(F) :: B, M, act, SA)|L]) \;\;\circ\!- \;\; verify(M) \;\& \\ \quad (event\_queue([]) \parr belief(F) \parr int\_stack([plan\_inst(Id, B, M, act, SA)|L])). \end{array}$$

If the previous rule cannot be applied, the current plan is suspended and a new event containing a *subgoal* trigger and a reference to the current plan is created:

$$\begin{array}{l} event\_queue([]) \parr int\_stack([plan\_inst(Id, achieve(F) :: B, M, act, SA)|L]) \;\circ\!- \\ \quad verify(M) \;\&\; not\_believed(F) \;\&\; (event\_queue([event(subgoal(F), Id))]) \parr \\ \quad int\_stack([plan\_inst(Id, B, M, susp, S)|L]). \end{array}$$

If the plan component is a *query* goal and F is a belief of the agent, then the query succeeds by using a rule similar to the first *achieve* rule (the plan instance at the top of the intention stack is a *query(F)* instead of an *achieve(F)*).

Plan termination and resuming. If a plan has been completed successfully, the *SuccActs* (sequence of internal actions) has to be executed. Furthermore, the awakening of a previously suspended plan may also be necessary. The following rules capture the execution of *SuccActs*:

$$\begin{aligned}&event\_queue([]) \parr int\_stack([plan\_inst(Id, [], \_, act, assert(F) :: SA)|L]) \multimapinv\\ &\quad not\_believed(F) \,\&\, (event\_queue([]) \parr belief(F) \parr\\ &\quad int\_stack([plan\_inst(Id, [], \_, act, SA)|L])).\end{aligned}$$

$$\begin{aligned}&event\_queue([]) \parr belief(F) \parr int\_stack([plan\_inst(Id, [], \_, act, retract(F) :: SA)|L])\\ &\quad \multimapinv \quad event\_queue([]) \parr int\_stack([plan\_inst(Id, [], \_, act, SA)|L]).\end{aligned}$$

When the execution of *SuccActs* has been completed, the plan instance can be popped off the intention stack, and the (suspended) lower plan instance can be awakened:

$$\begin{aligned}&event\_queue([]) \parr int\_stack([plan\_inst(\_, [], \_, \_, []), plan\_inst(Id, B, M, susp, SA)|L])\\ &\quad \multimapinv \quad event\_queue([]) \parr int\_stack([plan\_inst(Id, B, M, act, SA)|L]).\end{aligned}$$

When an intention stack becomes empty, it is removed using the rule

$$int\_stack([]) \multimapinv \bot.$$

As mentioned before, such a specification can be directly executed in a logic programming language by creating an initial set of events and intentions.

5.2.1.2 ProlAg implementation of a BDI architecture

By refining the $\mathcal{E}_{hhf}$ specification into a concrete ProlAg implementation, we aim at the realization of a software product based on more established technology, that can be used in an actual MAS prototype operating in a concrete application domain. We now present an engine for the BDI architecture which is derived directly from the previously defined $\mathcal{E}_{hhf}$ specification.

An engine for the BDI architecture. Mapping the $\mathcal{E}_{hhf}$ specification of BDI into a ProlAg program does not require great effort. In fact, the $\mathcal{E}_{hhf}$ data structure previously defined finds a direct mapping into ProlAg terms. Furthermore, the representation of *beliefs intentions* and *event queue* can be stored in the agent *state*, and *plans* can be included in the agent *program* as facts.

We note that the $\mathcal{E}_{hhf}$ clauses, used for specifying the BDI engine, assume the general form

$$G_1 \,\&\ldots\&\, G_m \;\Rightarrow (A_1 \parr \ldots \parr A_n \multimapinv G_{m+1} \,\&\ldots\&\, G_{m+k} \,\&\, A_{n+1} \parr \ldots \parr A_{n+h}),$$

where all the formulas $G_1, \ldots, G_{m+k}, A_1, \ldots, A_{n+h}$ are atomic. In order to translate a multi-conclusion guarded clause C of such a form, we introduce an auxiliary predicate p_C, defined as

$$\begin{array}{l} p_C \text{:- } retract\_state(A_1), \ldots, retract\_state(A_n),\ G_1,\ \ldots, G_{m+k}, \\ \quad assert\_state(A_{n+1}),\ \ldots, assert\_state(A_{n+h}). \end{array}$$

where *retract_state* and *assert_state* are the state update predicates described in Section 3.3.6. The execution of $retract\_state(A_1), \ldots, retract\_state(A_n)$ consumes the atomic formulas $A_1,\ \ldots, A_n$; the proof of $G_1,\ \ldots, G_{m+k}$ tests both the clause guard and the conditions over the current state, and the actions $assert\_state(A_{n+1}),\ \ldots, assert\_state(A_{n+h})$ add new information to the state itself.

Applying this transformation to every $\mathcal{E}_{hhf}$ clause, we can obtain a corresponding set of ProlAg clauses. Such a set can be partitioned by grouping together clauses regarding the four main activities performed by a BDI agent that are *perception*, *plan_triggering*, *plan_execution* and *action_execution*.

The implementation in ProlAg of the BDI engine is defined by a predicate `cycle`. The cycle defining the BDI engine is simply:

$$cycle \text{:- } perception; plan\_triggering; plan\_execution; action\_execution.$$

Obviously, more sophisticated strategies can be implemented, such that the execution of the four activities takes additional information into account.

Each activity can be defined by the clauses which belong to the corresponding activity group, for example

$$plan\_triggering : -p_{C_1};\ \ldots\ ; p_{C_s}.$$

where $p_{C_1}, \ldots, p_{C_s}$ are the clauses of the plan triggering group.

We do not present the complete code for the BDI engine here, but give only (part of) the rules for handling internal triggers and external actions. The rule for handling an internal trigger belongs to the definition of *plan_triggering* (as do the rules for external triggers) and is given by

$$\begin{array}{l} plan\_triggering \text{:- } retract\_state(mail\_box([event(T, Id1)|L]), \\ \quad retract\_state(int\_stack([plan\_instance(Id1, B_1, M_1, act, SA_1)|L_1])), \\ \quad plan(T, C, B, M, SA), internal(T),\ verify(C),\ get\_new\_id(Id), \\ \quad assert\_state(mail\_box([L]), \\ \quad assert\_state(int\_stack([plan\_instance(Id, B, M, act, SA), \\ \qquad plan\_instance(Id1, B_1, M_1, act, SA_1)|L_1])). \end{array}$$

Here each call to a p_{C_i} has been substituted by its definition. The goal $get\_new\_id(Id)$ creates a new identifier.

To execute a plan step involving an external action the engine uses the rule

$$\begin{array}{l} plan\_execution \text{ :-} \\ \quad retract\_state(int\_stack([plan\_instance(Id, [A|Body], Maint, active, SA)|L])), \\ \quad retract\_state(mail\_box([])),\ external(A),\ verify(Maint), \\ \quad assert\_state(int\_stack([plan\_instance(Id, Body, Maint, active, SA)|L])), \\ \quad assert\_state(mail\_box([])), assert\_state(execute(A)). \end{array}$$

The rule for executing an external action is

$$\begin{array}{ll} action\_execution \text{ :-} & retract\_state(execute(send(Receiver, Message))), \\ & send(Receiver, Message). \end{array}$$

where the verification of goal $send(Receiver, Message)$ has a side effect of performing the sending of the message. Note that this specialized rule can be used, because in ProlAg the sending of messages are the only external actions allowed.

5.2.1.3 Discussion

We have described a logic based approach to specify the BDI agent architecture. The language we propose is based on a linear extension of classical logic languages such as Prolog. Its execution mechanism is goal directed and is based on clause resolution and unification as is usual in logic programming.

There are some other approaches to the specification of the BDI architecture. In [25] a specification for a BDI architecture is given using **Z**, a specification language based on set theory, first-order logic, and the notions of state space and transformations between states. We think that linear logic may be a more natural candidate for the specification of systems which involve the notion of state, transformations, and resources. The main advantage of $\mathcal{E}_{hhf}$ over **Z** is the fact that it is directly *executable*. Like **Z**, it supports specification at different levels of abstraction like **Z**. Unlike **Z**, its higher-order extensions (**Z** is first-order) greatly facilitate meta-programming.

Another formal specification for the BDI architecture is presented in [62]. It is based on a particular temporal logic language, Concurrent MetateM, which supports the specification of concurrent and distributed entities. Goals, beliefs and plans are represented by means of formulas in temporal logic, which are executed by a run-time execution algorithm. Verification of the specifications is rather direct. $\mathcal{E}_{hhf}$, like Concurrent MetateM, can easily support broadcast message-passing and can simulate meta-level activity by means of its higher-order features.

5.3 HEMASL

Agent-based software applications need to be able to incorporate agents with heterogeneous architectures in order that each agent is able to optimally perform its task. Many existing specification languages for agents commit to a particular architecture, thus making their adoption difficult when different agent architectures, ranging from reactive to rationale ones, are involved. A more flexible language, coping with the heterogeneity of agent architectures, can prove useful in many contexts.

We present HEMASL (*HEterogeneous Multi-Agent Systems Language*), a simple meta-language used for specifying intelligent agents and multi-agent systems when different and heterogeneous agent architectures must be catered for. This language has been introduced in [53]. HEMASL is much closer to widespread existing specification and implementation languages than logic languages. It can be considered as a first step towards defining an "intermediate" language which could make it much easier to animate and incorporate traditional specification languages into CaseLP. The integration of HEMASL into CaseLP is currently under study and is based on the translation of HEMASL specifications into $\mathcal{E}_{hhf}$ or ProlAg clauses. We argue that adding HEMASL to CaseLP, can enhance its flexibility and usability.

This section describes some of the features of the language, presents examples of its use, outlines its operational semantics, and delineates how an HEMASL specification may be mapped into object oriented languages such as Java. A library of Java architectures can be used as the basis for implementing agent-based applications.

5.3.1 Basic features of the language

The basic features of HEMASL make it suitable for agent architecture specification and for the incorporation of heterogeneous agents into the same MAS. The important characteristics of HEMASL are presented below.

Agent model. The agent model supported by HEMASL is an abstraction of many existing architectures. This facilitates the development of the architecture specification. Moreover, since this model is the same as the one we adopted for developing agents in CaseLP, it allows for the direct integration of HEMASL specifications into CaseLP.

Hierarchy of abstraction levels. HEMASL provides constructs for specifying a MAS at four different levels of abstraction. It supports a modular and flexible representation of the system based on the concepts of abstract and concrete architecture, agent class and agent instance. These different abstraction levels help the designer to model the heterogeneity of agent services and architectures.

Situatedness and social ability. The ability of an agent to interact with the surrounding environment consists of its ability to perceive events which take place in the environment and to perform actions which modify the environment. HEMASL provides an explicit model of the environment as well as primitives that can be used by agents to sense and modify the environment. It also provides constructs for modelling the exchange of messages between agents.

Operational semantics. The operational semantics of HEMASL is based on the concepts of "MAS configuration" and "configuration transition". It is given defining an abstract interpreter for the language, which can be used to animate specifications.

In the following sections we expand the concepts outlined above. Section 5.3.2 outlines the computational model of our agents and describes the hierarchy of abstraction levels we use for describing them. Section 5.3.3 demonstrates the main features of HEMASL through example specifications of two agent architectures along with a MAS made up of these agent. Section 5.3.4 outlines the operational semantics of HEMASL. Section 5.3.5 discussed the mapping between HEMASL and Java. Section 5.3.6 compares our approach to other proposals.

5.3.2 Agent model and hierarchical structure of a MAS

The agent model we assume is the same as that presented in Section 2.3.3. According to this model an agent is characterized by its *state*, *program*, and *engine*. In this chapter we use the BDI architecture[3] as a running example for illustrating the potentiality of HEMASL. The state of a BDI agent contains its beliefs, goals and intentions. The program of a BDI agent is determined by its plans. A typical BDI engine is characterized by the following loop:

1. perceive an event,
2. identify a set of plans which can be used to manage the perceived event according to the current beliefs,
3. select one of the plans from the set,
4. add the selected plan to the intention set,
5. select one intention and execute one instruction from the selected intention,
6. drop successful goals. and

[3]See Section 2.3.4.1 of this dissertation for a description of the BDI agent architecture.

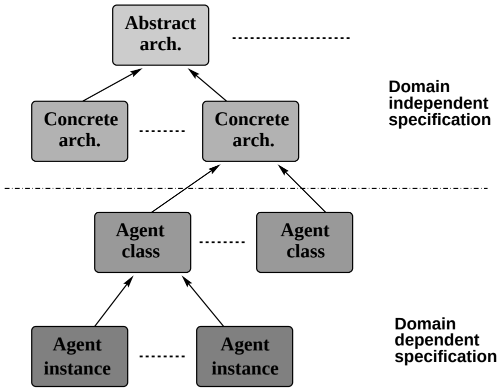

Figure 5.1: Abstraction hierarchy.

7. go to step 1.

The architecture of an agent is characterized by the components which contain its state, its program, and an engine which operates upon them. The content of the agent components will be expressed using some architecture dependent object language for which the engine provides an interpreter. Here, we do not commit to any particular object language.

A specification language for a heterogeneous MAS should ensure modularity and flexibility in the definition of agent architectures. To provide such modularity and flexibility, we introduce a four level abstraction hierarchy, illustrated in Fig 5.1.

5.3.2.1 Abstract architecture

The abstract architecture defines the components in the architecture and the basic structure of the engine. It is possible to provide guidelines for the realization of any "macro-instruction" (procedure) present in the engine, without necessarily giving all the implementation details.

Intuitively speaking, an abstract architecture defines an agent's data structures and the engine that organizes an agent's internal activities, without going into too many low-level details. Examples of irrelevant details at this level would be the way data structures are implemented or how an intention is selected in a BDI architecture.

5.3.2.2 Concrete architecture

An abstract architecture may give rise to several concrete architectures. At this level of abstraction, the data structures used for the various architecture components are chosen as is the functioning of the agent. A concrete architecture is defined by starting from an abstract architecture as follows:

- each component is assigned a type, chosen among these the language provides;
- each macro-instruction of the engine is implemented.

5.3.2.3 Agent class

The same concrete architecture can be used to build agents that work in different application domains. Domain dependent behavioural patterns are added by defining the agent program at the class level. A class is defined by instantiating the components in the concrete architecture, which contain the program of the agent.

5.3.2.4 Agent instance

A MAS may require several instances of the same agent class to work concurrently by starting from different initial states. This is captured in our hierarchy by defining an agent instance level in which, starting from an existing class, architecture components containing the state are initialized.

5.3.2.5 MAS environment and ether

Besides its internal representation, an agent is also characterized by its ability to perceive the surrounding environment and to interact with other agents within the system. In HEMASL the *environment* (*env*) is modelled as a collection of facts representing those features of the physical environment which are relevant to the application domain. Agents are able to perceive the environment directly, but they can only modify it by interacting with an "environment agent".

Suppose that an agent *robot* is able to perform the action `turn_the_engine_on`. There are several consequences of this action on the environment (for example, an increase in noise, temperature and pollution). It is not realistic that the robot would know all the consequences of its actions, thus it can not directly modify all the environment state information itself. Therefore we employ a domain-dependent "environment agent", whose task is to evaluate the consequences of primitive agent actions, and to update the environment consequently.

As far as communication is concerned, an agent interacts with the "environment agent" and with other agents by means of the *ether* (*eth*), a data structure to which messages are submitted, and from which they are retrieved. HEMASL provides primitives for communication.

5.3.3 Syntax and Examples of Use

5.3.3.1 Primitive Instructions and Statements

HEMASL provides primitive instructions for managing the information contained in the architecture components, for delivering and receiving messages, for perceiving events within the environment, for generating events to modify it, and for performing actions upon it. These basic instructions can be composed using the statements: *variable declaration* and *assignment*; *procedure call*; *deterministic choice* (**if-then-else**); *loop* (**while**); *concatenation* (**;**); *non deterministic choice* ($|$); *concurrent execution* ($||$).

The instructions which operate on the agent's internal components are

- **get_comp**(c, m, v),
- **put_comp**(c, m, e), and
- **del_comp**(c, m)

where c stands for the name of the component, v is a variable, e is an expression and m is the mode of insertion, extraction or deletion of an element into/from a component. The mode m depends on the component type. For example, if the component is a list, m can assume `head` and `tail` as values.

The instructions for message exchange are

- **send**(r, m),
- **rec**(s, m), and
- **block_rec**(s, m)

where r, m, and s represent the receiver, the message, and the sender respectively. When a blocking reception is performed, execution of the agent program is blocked until the named agent s sends a message. Conversely, in the case of a non-blocking reception, execution will continue even if no message sent by s can be retrieved from the ether.

The instructions for the perception and generation of an event e in the MAS environment are

- **perceive**(e),
- **put_event**(e), and
- **remove_event**(e).

Insertion and deletion of an event into/from the environment are reserved operations, which can be executed only by the "environment agent". Perception operations can be performed by all the agents in the system.

Finally, any agent can execute an action a using the primitive

- **exec**(a).

The effect of this primitive is to send a message with content a to the "environment agent". This agent evaluates the consequences of performing a in the environment and modifies the environment by means of **put_event**(e) and **remove_event**(e) primitives.

5.3.3.2 Abstract Architecture Definition

To define an abstract architecture, we declare its components, its engine and the procedures of this engine. As an example, consider a BDI-like architecture. As previously described, the architecture is characterized by four components: beliefs, goals, intentions, and plans.

The definition of an abstract BDI architecture is depicted in Figure 5.2. The keyword **class** means that the *plans_component* will be instantiated during the definition of the agent class, and thus the data contained in it will represent the *program* of the agent. The keyword **agent** suggests that the *beliefs_component*, *goals_component* and *intentions_component* will contain information representing the agent's state.

The engine consists of an endless "while" loop which execute a sequence of macro-instructions defined as procedure calls. *Global variable* declarations can be included in the engine body. The body of a procedure can be left only partially specified in the abstract architecture. In this case, the implementation details of the macro-instruction defined by the procedure must be completely described when defining the concrete architecture. A macro-instruction can also be left completely undefined.

In this example, *perceive_event()* is not defined at all, while *upgrade_intentions_component()* is partially defined, as depicted in Figure 5.3. The body of *upgrade_intentions_component()* contains first declarations of *local variables*. The procedures *is_empty_selected_plans(!empty)*, *select_plan(!plan)*[4], and *put_intentions_component(***gettupla***(plan,3))* are not defined at the abstract architecture level.

[4]The symbol "!" in the procedure call means that the argument is passed by reference.

```
abstract_architecture {bdi} {
   components {
      class plans_component;
      agent beliefs_component, goals_component, intentions_component;
   };
   procedures
     { ... definition of the engine's procedures ... }
   engine {
      decl event, selected_event, triggered_plans, selected_plans;
      while true do
        perceive_event();
        plan_triggering();
        plan_selection();
        upgrade_intentions_component();
        exec_intention();
        drop_succesful_goals();
      endwhile
   }
}
```

Figure 5.2: BDI abstract architecture.

5.3.3.3 Concrete Architecture Definition

During the definition of the concrete architecture, all the components are assigned a type, the global variables are initialized, and the definitions of partially specified procedures are completed. To illustrate two different implementation choices, we consider two concrete BDI architectures, bdi_1 and bdi_2, obtained from the previously defined abstract BDI architecture *bdi*. The implementation of the BDI concrete architectures is depicted in Figure 5.4 and 5.5.

In the concrete architecture bdi_1, the *plans_component* has type `stack`, the *beliefs_component* has type `set`, the *goals_component* has type `set`, and the *intentions_component* has type `queue`. External events are either events generated by the environment or messages sent by other agents in the system. An agent implemented using this architecture will behave both reactively (to changes in the environment) and socially (by sending and receiving messages).

In bdi_2, the *plans_component*, the *beliefs_component*, and the *goals_component* are `sets`, while the *intentions_component* is a `stack`[5]. The only perceived events are those generated

[5]These types are probably not the most reasonable to assign to BDI components. They have been chosen to demonstrate language flexibility.

```
upgrade_intentions_component() {
    decl plan, empty;
    is_empty_selected_plans(!empty);
    if not(empty) then
        select_plan(!plan);
        put_intentions_component(gettupla(plan,3))
    else skip
}
```

Figure 5.3: A partially defined procedure.

by the environment. This architecture gives rise to strongly reactive agents with no the ability to receive messages.

In both concrete architectures, perceived events are collected in the global variable *event* which has type `queue`. In bdi_1, an event is perceived from the environment by means of the **perceive**(e_1) procedure. The global variable *event* is updated by inserting the perceived event into it. A message from *sender* is received in parallel with the perception of the environment, and the received message is also put into the event queue. In bdi_2, only events taking place in the environment are perceived and inserted into the event queue.

5.3.3.4 Definition of Agent Classes and Instances

After the concrete architectures have been defined, the MAS can be instantiated (Figure 5.6). Firstly, the classes of agents are defined. For each agent class, one of the previously created concrete architecture is chosen. Then, the primitive **init_comp** is used to initialize the architecture components that contain the agent program. We could, for example, specify a MAS in which there were three agent classes: two of them, ac_1 and ac_2 say having a bdi_1 concrete architecture, while the other one ac_3 had a bdi_2 architecture. The three architectures would have different programs, composed of domain dependent plans.

To start a simulation we have also to initialize the environment *env* and the ether *eth*, using the primitives **init_ENV** and **init_ETH**. The initial environment and ether will contain, respectively, a set of events $event_1, \ldots, event_e$ and a set of messages $msg_1, \ldots, msg_m$.

Finally, the individual agent instances are created by using the primitive **create_agent** as many times as there are agents instances in the MAS. For example, we could create a MAS composed of two instances of ac_1, one istance of ac_2 and one instance of ac_3. For each agent instance, the initial internal state is also set by using the primitive **init_comp**.

```
architecture {bdi_1} is a {bdi} {
  components {
     plans_component: stack;
     belief_component, goals_component: set;
     intention_component: queue
  };
  init_global_vars { ...};
  procedures {
     perceive_event() {
        decl sender, e_1, e_2;
        ( perceive(e_1); event := insqueue(event, e_1)) ||
        ( get_belief_component("sender", !sender); rec(sender, e_2);
        event := insqueue(event, e_2) )
     };
     ...
  }
}
```

Figure 5.4: Definition of concrete architecture bdi_1.

5.3.4 Operational Semantics

The operational semantics of HEMASL is given by a tree representing the transitions between tuples of *MAS configurations*. A full account of the semantics can be found in [51], and only an outline is given here. A MAS configuration has the form

$$[env,\ eth,\ s_{a_1},\ \ldots,\ s_{a_n},\ s_{env\_agt}]$$

where env and eth represent the state of the MAS environment and the state of the MAS ether respectively, $s_{a_1},\ \ldots,\ s_{a_n}$ represent the states of the "common" agents and s_{env_agt} represents the state of the "environment agent".

5.3.4.1 The environment and the ether

The environment env is a set of pairs $\langle fact,\ value\rangle$, where $fact$ is a string representing a relevant fact for characterizing the MAS environment (for example, "temperature"), and $value$ is a string representing the current value of the observed fact (for example, "25"). The ether eth is a set of triples of strings $\langle sender,\ receiver,\ content\rangle$. The term *sender* must be instantiated with the name of the sender and *receiver* may be instantiated with an agent name or with the string "all", which models broadcast communication. The ether

architecture $\{bdi_2\}$ **is a** {bdi} {
  components {
    plans_component, belief_component, goals_component: `set`;
    intention_component: `stack`
  };
  init_global_vars { ...};
  procedures {
    perceive_event() {
      decl e;
      perceive(e); event := **insqueue**(event, e)
    };
    ...
  };
}

Figure 5.5: Definition of concrete architecture bdi_2.

contains all the messages which have been delivered but are yet to be received by the reception agent.

5.3.4.2 Agent state

The state of the agent a is a pair $\langle ex\_env_a,\ cmp_a \rangle$. It contains information about the architecture components content (cmp_a) and about the execution environment of the agent engine (ex_env_a)[6]. The function ex_env_a associates values to variables and local execution environments to procedure identifiers. This is in coherence with the meaning assigned to the execution environment in the imperative languages semantics.

5.3.4.3 MAS execution

The relation $\overset{MAS}{\longmapsto}$ defines the transitions between MAS configurations. The actions of the "environment agent" may affect its state, the ether and the MAS environment, and thus the relation $\overset{env\_agt}{\longmapsto}$ is defined over *environment agent configurations* of the form $(s_{env\_agt},\ eth,\ env)$. Conversely, the actions of a "common agent" a cannot directly modify the MAS environment, and thus $\underset{a}{\overset{agt}{\longmapsto}}$ is defined over *agent configurations* $(s_a,\ eth)$. Let $\mathcal{A}$ be the set of names of the agents in the MAS. The above concepts are formalized by the

[6]Some confusion could arise due to the presence of both the agents' execution environment ex_env_a and the MAS environment env. The context always clarifies the meaning of the term "environment".

MAS {
  class_agent $\{arch\_name_1,\ classagent\_name_1\}$ {
    init_comp $comp\_name_1\ [elem_{1_1},\ \ldots,\ elem_{1_i}];\ \ldots$
  };
  $\vdots$
  init_ENV $[event_1,\ \ldots,\ event_e]$;
  init_ETH $[msg_1,\ \ldots,\ msg_m]$;
  create_agent$(classagent\_name_{t_1},\ agent\_name_1)$ {
    init_comp $comp\_name_1\ [elem_{1_1},\ \ldots,\ elem_{1_i}];\ \ldots$
  };
  $\vdots$
}

Figure 5.6: MAS definition schema.

meta-rules

$$\frac{(s_{env\_agt},\ eth,\ env) \overset{env\_agt}{\longmapsto} (s'_{env\_agt},\ eth',\ env')}{[env,\ eth,\ S,\ s_{env\_agt}] \overset{MAS}{\longmapsto} [env',\ eth',\ S,\ s'_{env\_agt}]}$$

$$\frac{(s_a,\ eth) \underset{a}{\overset{agt}{\longmapsto}} (s'_a,\ eth')}{[env,\ eth,\ S,\ s_{env\_agt}] \overset{MAS}{\longmapsto} [env,\ eth',\ S',\ s_{env\_agt}]} \quad a \in \mathcal{A}$$

where S' is obtained by substituting s_a with s'_a in S.

5.3.4.4 Agent execution

To give an overview of the language semantics we will describe some meta-rules governing the execution of "common" agents. The meta-rules for the execution of the "environment agent" are similar to those of the "common" agents, but also include semantic definition of **put_event** and **remove_event**. The relation $\underset{a}{\overset{agt}{\longmapsto}}$ is defined by the meta-rule

$$\frac{(\langle ex\_env_a, cmp_a\rangle,\ eth) \underset{a,i}{\overset{ins}{\longmapsto}} (\langle ex\_env'_a, cmp'_a\rangle,\ eth')}{(\langle ex\_env_a, cmp_a\rangle,\ eth) \underset{a}{\overset{agt}{\longmapsto}} (\langle ex\_env'_a, cmp'_a\rangle,\ eth')} \quad \begin{array}{l} a \in \mathcal{A} \\ N\_I(a) = i \end{array}$$

The function $N\_I(a)$ returns the next instruction which the agent a must execute. The relation $\underset{a,i}{\overset{ins}{\longmapsto}}$ is defined over agent configurations and depends both on the agent performing the instruction and on the instruction itself. We will now define this relation for some of the HEMASL basic constructs.

5.3.4.4.1 Event perception

$$\frac{f \underset{ex\_env_a}{\overset{e\_exp}{\longmapsto}} f'}{(\langle ex\_env_a, cmp_a\rangle,\ eth) \underset{a,\mathbf{perceive}((f,x))}{\overset{ins}{\longmapsto}} (\langle ex\_env_a[v/x],\ cmp_a\rangle,\ eth)} \quad \begin{array}{l} a \in \mathcal{A} \\ (f', v) \in env \\ x \in\ d(ex\_env_a) \end{array}$$

The relation $\underset{ex\_env_a}{\overset{e\_exp}{\longmapsto}}$ evaluates an expression according to the current execution environment of agent a. The function $d(ex\_env_a)$ returns the domain of the function ex_env_a and $ex\_env_a[v/x]$ is obtained by composing ex_env_a with the function which associates v to x. The effect of a perception instruction is to modify the agent's state by creating a new association between the variable argument of **perceive** and the value associated to the perceived fact.

5.3.4.4.2 Message delivery

$$\frac{r \underset{ex\_env_a}{\overset{e\_exp}{\longmapsto}} r' \qquad c \underset{ex\_env_a}{\overset{e\_exp}{\longmapsto}} c'}{(\langle ex\_env_a, cmp_a\rangle,\ eth) \underset{a,\mathbf{send}(r,c)}{\overset{ins}{\longmapsto}} (\langle ex\_env_a, cmp_a\rangle,\ eth \cup \{(a,\ r',\ c')\})} \quad a \in \mathcal{A}$$

The effect of a **send** is the insertion of a new triple into the ether.

5.3.4.4.3 Message reception

$$\frac{s \underset{ex\_env_a}{\overset{e\_exp}{\longmapsto}} s'}{(\langle ex\_env_a, c_a\rangle,\ eth) \underset{a,\mathbf{rec}(s,x)}{\overset{ins}{\longmapsto}} (\langle ex\_env_a[c/x], c_a\rangle,\ eth/\{(s',\ a,\ c)\})} \quad \begin{array}{l} a \in \mathcal{A} \\ (s',\ a,\ c) \in eth \\ x \in\ d(ex\_env_a) \\ a \neq\ \text{“}all\text{”} \end{array}$$

This meta-rule can be applied when the receiver field of the message is a particular contains the name of a particular agent and not the string "all". The effect of a **rec** is to associate the content of the received message to the variable argument of **rec** and to remove the read message from the ether. If the receiver field is set to "all", the message is not removed from the ether until all the agents in the system have read it[7].

The same rules are also given for performing the blocking reception. The difference between the semantics of blocking and non-blocking message reception is that for a non-blocking reception there is an additional meta-rule. This rule is applied when the desired message is not present in the ether. In this case the effect of a **rec** is to create an association between the variable argument of **rec** and the string "null", with no side-effects on the ether. The semantics of the blocking reception is undefined if the message is not present in the ether and will thus force the agent to block its execution.

[7]The ether makes sure that the receivers of a broadcast message read the message only once, and that the message is removed once all the recipients have read it.

5.3.4.4.4 Action execution

$$\frac{act \overset{e\_exp}{\underset{ex\_env_a}{\longmapsto}} act'}{(\langle ex\_env_a, cmp_a\rangle,\ eth) \overset{ins}{\underset{a,\mathbf{exec}(act)}{\longmapsto}} (\langle ex\_env_a, cmp_a\rangle,\ eth \cup \{\ (a,\ env\_ag,\ act')\ \}} \quad a \in \mathcal{A}$$

The semantics of an **exec** instruction is to send a message to the "environment agent", which contains the action to be performed. This is achieved by modifying the ether.

5.3.5 Towards an OO implementation of HEMASL specifications

We think that HEMASL could be used as a specification language for the development of agents which will be implemented into commercial, object-oriented programming languages. In fact, HEMASL specifications are described in terms of a hierarchy of abstraction levels, which makes their implementation in an object oriented language very appealing. In this section we present some ideas on how HEMASL specifications should be implemented in Java [52]. The outlined translation of HEMASL specifications into Java programs represents a first step towards a real implementation of a system.

Some considerations guide our mapping. Firstly, any agent is an autonomously executing entity, with its own thread of execution. This observations naturally leads to the implementation of agents as concurrently executing Java *Threads*. In HEMASL an abstract architecture describes the architecture components and the "macro-instructions" of the engine, possibly providing implementations for some of them. An entity with these features finds a natural mapping to an `abstract class` in Java. This class can be made to extend he *Thread* class of Java, and some methods of this class can be left unimplemented. As far as a HEMASL concrete architecture is concerned, its aim is to complete the information provided in the abstract architecture, by defining those methods for which only the signature was given and by assigning a type to each architecture component. This HEMASL entity can be naturally mapped into a Java `class` which `extends` the Java abstract architecture. Finally, in a HEMASL agent instance both the program and the state components are filled with the proper information. Such an entity can be implemented as a Java object. These considerations lead to the following implementation of HEMASL entities.

5.3.5.1 HEMASL basic types

HEMASL provides the following basic types: *List, Queue, Set, Stack* and *Tuple*. We assume the existence of a Java class implementing each HEMASL type. All these classes inherit from a common Java superclass called *Hemasl_type*.

```
abstract class Bdi extends Thread {
        abstract Hemasl_type get_plans_component();
        abstract void set_plans_component(Hemasl_type plans);
        abstract Hemasl_type get_beliefs_component();
        abstract void set_beliefs_component(Hemasl_type plans);
        ...

        ...Auxiliary (abstract) methods...;

        abstract void percept_event();
        public void plan_triggering()
            { ...implementation of the method... };
        abstract void plan_selection();
        ...;

    public void run() {
        while (true) {
            percept_event();
            plan_triggering();
            plan_selection();
            upgrade_intentions_component();
            exec_intention() }
    }
};
```

Figure 5.7: BDI abstract architecture in Java.

5.3.5.2 Abstract architecture

A HEMASL abstract architecture HAA can be implemented as a Java abstract class (JAA, *Java Abstract Architecture*) that extends the class *Thread*. The architecture components are implicitly declared by means of the methods for reading and writing them, which are declared `abstract`[8]. Also the procedures left undefined in the HAA are implemented as `abstract` methods. The engine of the HAA can be implemented by overriding the method *run* of *Thread*.

Figure 5.7 sketches the mapping from HEMASL to Java for the abstract BDI architecture depicted in Section 5.3.3.

[8] There are technical reasons for delaying the explicit declaration of the components when implementing the concrete architecture. Here we do not discuss these subtle implementation details.

```
class Bdi1 extends Bdi {
        Stack plans_component;
        Set beliefs_component;
        Set goals_component;
        Queue intentions_component;

        public void percept_event()
            { ... implementation of the method ... };
        ...;

        public Bdi1(
            Stack plans, Set beliefs, Set goals, Queue intentions)
            { implementation of the constructor };
    }
```

Figure 5.8: BDI concrete architecture in Java.

5.3.5.3 Concrete architecture

A HEMASL concrete architecture HCA can be implemented as a Java concrete class (JCA, *Java Concrete Architecture*) by extending the JAA which implements the corresponding HAA. In the JCA each component is declared explicitly and is given a precise type form among those supported by HEMASL. The constructor for JCA objects is also defined at this stage. It takes the architecture components as arguments, thus allowing the creation of an agent instance.

In Figure 5.8 the definition of the concrete architecture $bdi1$ is outlined.

5.3.5.4 Agent class

HEMASL classes do not have a direct counterpart in Java. Their implementation in Java is realized by defining different Java objects to characterize the program components of the different classes. Agent instances belonging to the same agent class will have program components referring to the same objects, which will be defined only once for each agent class.

As an example, assume that we want to build a $bdi1$ **diagnostic_agent** class and a $bdi1$ **robot_agent** class. We need to define the proper plans for both the classes, as depicted in Figure 5.9. In this simple example we have just one "program component", namely the *plans_component*. In general, there could be more than one component, and therefore we

```
Stack diagnostic_agents_class_plans =
    new Stack(arguments characterizing
    the diagnostic agent program);

Stack robot_agents_class_plans =
    new Stack(arguments characterizing
    the robot agent program);
```

Figure 5.9: Programs for BDI diagnostic and robot agents in Java.

```
Bdi1 diagnostic_agents_instance1 = new Bdi1(
  diagnostic_agents_class_plans,
  new Set(...), new Set(...), new Queue(...));

Bdi1 diagnostic_agents_instance2 = new Bdi1(
  diagnostic_agents_class_plans,
  new Set(...), new Set(...), new Queue(...))

Bdi1 robot_agents_instance1 = new Bdi1(
  robot_agents_class_plans,
  new Set(...), new Set(...), Queue(...));
```

Figure 5.10: Instances of BDI diagnostic and robot agents in Java.

should define all of them at this stage.

5.3.5.5 Agent instance

A HEMASL agent instance HAI is obtained from an agent class HAC by filling the components that correspond to the agent state. In our mapping a *Java Agent Instance*, JAI, is an object whose *program* components are shared with all the agent instances of the same class and whose *state* components are "individual".

In Figure 5.10 two instances of a diagnostic agent and one instance of a robot agent are defined.

5.3.6 Comparison

In this section we compare our approach for specifying heterogeneous agent architectures with other proposals: [29], by Fisher; [36], by Hindriks et al.; [8], by Treur et al., [32] by De Giacomo et al.. The proposals are compared according to their capabilities to specify architectures complying with the model of the agent presented in Section 2.3.3 (i.e., in terms of state, program and engine).

In [29], Fisher presents the specification of an "abstract agent architecture" using Concurrent MetateM, a specification language based on temporal logics. The architecture model allows the designer to incorporate into the design different kinds of *behaviour*, to be performed by *groups* of sub-agents. Each behaviour is described by a set of temporal logic rules that specify how the future state of an agent computation are obtained from its present state. Examples are presented regarding the specification of *reactive*, *deliberative*, and *social* behaviour and on the composition of these primitive types of behaviour into different kinds of layered architectures. This approach seems highly suitable for representing the agent program. Temporal logic is very expressive for representing possible courses of agent action. Moreover, modularization (provided by grouping) allows for the clear expression of different kinds of agent behaviour. On the other hand, this approach does not give explicit representation of the state and the engine of the agent. Neither the specification of the agent engine nor the representation of the agent state are included in the abstract agent architecture. Instead, they are implicitly maintained in the *interpreter* for Concurrent MetateM, that executes Concurrent MetateM specifications. The interpreter operates along two dimensions. Firstly, it tries to satisfy future states of agent computations as soon as possible, according to the temporal rules that specify the agent behaviour. Secondly, it has to implement some control strategy that organizes the activities of the various types of behaviour.

Hindriks et al. follow a different approach. They focus mainly on the specification of agent engines to be used as meta-interpreters for many different object level languages used for agent programs. They assume that agent computations may be well expressed using programming languages based on the concepts of *beliefs*, *goals* and *rules* and define an imperative meta language that is used to describe a "standard" engine cycle which includes *sensing*, *rule application*, and *goal execution*. They also present a comparison between the *rule application* and *goal execution* phases of AGENT-0 [75], AgentSpeak(L) [73] and 3APL [36]. They argue that a *glass box* approach for defining agent engines which makes internal functioning of engines visible to the MAS developer, is the "right" approach. In such a way the developer can directly write programs which control the agent's internal activities. Following the approach by Hindriks et al., the development of agent engines is natural and immediate. This approach, however, limits the agent state to being composed of only beliefs and goals, and the agent program to being composed only by rules. Even if most of the agent architectures are designed in terms of these abstractions, the approach could not be general enough to fit some agent applications.

The DESIRE research focuses on compositional MAS and their application for handling complex tasks, as well as development methods for creating these systems. The structure of DESIRE specifications is based on the notion of a compositional architecture: an architecture made up of components with hierarchical relations among them. This approach flexibly supports the definition of different heterogeneous architectures. Some architecture components may contain the agent's state while others the agent's program. The diverse aims of these two kinds of components is not expressed, however, either from a syntactical or from a semantical point of view. The flow of information among the components is described by the specification of communication links. There is no engine governing the execution flow within the components of the agent. Even if the DESIRE framework does not provide explicit support for defining agent architectures in terms of state, program and engine, it has been successfully adopted to develop a library of heterogeneous architectures ranging from reactive and proactive ones, to reflective and BDI ones.

ConGolog specifies an agent's behaviour based on the actions the agent can execute. The language adopted for this purpose is *situation calculus*, a first-order language (with some second order features) used for representing dynamic domains. ConGolog includes facilities for prioritizing the execution of concurrent processes, interrupting execution when certain conditions become true, and dealing with external actions. These features make adoption of the language for implementing reactive agents possible. The ascription of mental attitudes to agents can be easily achieved by adapting the possible worlds model of knowledge to situation calculus. Communicative capabilities can also be modelled in the framework. ConGolog provides the means for describing both the declarative and the procedural knowledge of agents with capabilities ranging from reactivity to rationality. Nevertheless, all this must be done at the "program" level. The engine for these programs is always the same, as is the agent architecture.

HEMASL lets the architecture designer choose the most suitable components and agent programming languages, thus allowing for a great variety in the representation of the agent state and program, and for a high degree of flexibility in the architecture. Furthermore, the HEMASL primitives allow the architecture designer to build the engine as she/he wants it. We argue that an "opaque box" approach is the best solution for agent-based software development. The MAS developer should know the main characteristics of the architectures she/he can use for the applications, but she/he is not usually an expert and does not need to be burdened with implementation details regarding architecture control. By giving a great deal of freedom in architecture development, we can obtain a library including architectures with several data structures and control flows. In such a way, the MAS developer can (hopefully) find all the architectures to fits her/his needs in the library.

Chapter 6

CaseLP at work: KICKER

The second goal of our research work was to demonstrate that the method and tools incorporated into CaseLP can be profitably used to realize prototypes of real-world software applications. The case-study presented in this chapter concerns the realization of KICKER, a prototype for a decision support system to be used by European traffic operators in the management of freight train movements between inter-modal centres and yards. The KICKER project is a collaboration between the Information Systems Division of Italian Railways (FS S.p.A.) and the Computer Science Department (DISI) at the University of Genova, Italy. Some preliminary results of the project are presented in [19] and [20].

KICKER is based on the F-TTM module of the European project EuROPE-TRIS, [16], which realized a set of demonstration tools to provide traffic operators with several functions for managing the contingent scheduling of trains. These tools did not have any decision-making capabilities, so decisions regarding the creation and dispatching of new trains were left completely up to human operators. KICKER embeds the functions of the F-TTM tools into a unique system. This is done emulating the F-TTM tools, adding decisional knowledge to some of them and realizing new system components that co-ordinate the needed decisional processes.

The operative scenario on which KICKER is supposed to operate is the railway line between Bern (Switzerland) and La Spezia (Italy). This line is particularly interesting because it has been identified by European railways as a "freightway" from the north to the south of Europe[1].

This chapter is structured as follows. Firstly, we introduce the problem of freight train traffic management and outline the EuROPE-TRIS project. Then, in Section 6.2 we describe the operative scenario within which KICKER is supposed to operate, focusing in particular

[1]Freightways are railway lines heavily affected by freight train traffic and connected to inter-modal centres for exchange of goods.

on the decisions that the prototype must support. Sections 6.3 and 6.4 present a revised version of the F-TTM tools, that are used for the realization of KICKER, and also focus on the modelling of the decision processes. In Section 6.5 we describe in details the prototyping of KICKER in CaseLP and finally in Section 6.6 we consider the goals that have been achieved so far in the project.

6.1 Freight train traffic management

Train traffic has been dramatically increasing during the last few years, mainly because more and more people and goods have to travel or be transported through larger and larger areas. This is particularly true in Europe, where there is a high density of population and many industrial and commercial centres. The increasing need for mobility of goods and people has to be faced by railway companies in order to provide their costumers, either travellers or commercial/industrial companies, with a satisfactory level of service. In providing their services, railway companies have to take into consideration such factor as speed of transportation, safety and comfort of passengers, and flexibility in goods dispatching and collecting. In particular, flexibility is the main issue when inter-modal transportation systems are considered. In this case several different organizations (e.g., railways, trucking or bus-line companies, harbour authorities) are involved and each of them imposes its own set of demands and constraints. The fundamental aim is to make train transportation economically advantageous for all the parties involved.

European railway companies have been making a great effort to improve their services, through the massive adoption of computer technology. In the last few years they have been working to modernize their organizations, especially by introducing an increasing level of *automation*, for example in the processes related to train circulation and signalling, seat reservation and customer support. This growing adoption of technology is also enabling processes to be harmonized between European railway companies.

The harmonization of freight transportation is still at an earlier stage and is an urgent priority. One main issue is the organization of an efficient system for inter-modal freight transportation. Such a system would allow goods to be dispatched, by means of railway and road carriers, from European harbours to their final destinations and vice versa, without too many bureaucratic and technological obstacles. In Europe in particular, there is an increasing demand for the short-term scheduling of freight trains. Short-term scheduling means that unscheduled trains are dispatched on demand along the same railway lines where scheduled trains normally run. Short-term scheduling is particularly necessary if the number of trains running on a railway line and the availability of tracks along the line are only known on a day-by-day basis. Even though long term scheduling of train traffic is certainly easier to plan, it does not allow sufficient flexibility for meeting customer demand.

If railway companies want to be competitive with respect to other form of transportation, they must improve the short-term scheduling process for trains.

In order to improve their services in the freight market, European railway companies have introduced the concept of freightway, defined as a European railway corridor along which the freight train traffic is optimized in terms of speed and cost.

New automatic decision support tools need to be provided to freight traffic operators, in order to help them in making optimal decisions quickly. The availability of these kind of tools should dramatically reduce decision time and simplify inherent procedures. Human traffic operators will no longer be burdened of all of decision making responsibility. This should certainly help in the full utilization of freightways, which are at the moment little more than a theoretical concept. In particular, automatic systems supporting short-term scheduling decisions would be very useful for maximizing the availability of tracks and optimizing traffic flow.

6.2 The KICKER project

Due to the complexity of the problem, automatic tools for the organization of goods distribution need to be based on sophisticated technologies and need to provide a wide range of operative solutions. The KICKER project is a contribution to the applied research on "intelligent" systems that will support decision making in the field of inter-modal freight transportation.

KICKER aims at the realization of a prototype for a decision support system to be used by European traffic operators, mainly for short-term management of freight train traffic. This activity involves decisions about the *allocation of the necessary resources*, the *granting of all the needed authorizations* and the *dispatching management* of circulating trains. Key factors that have to be taken into account in making these decisions are the transportation of *dangerous goods* and the *management of yards* along railway lines. The choice of an international freightway as an operative scenario allows us to model the prototype to include decision processes concerned with trans-national transportation.

The aim of KICKER is to create an instrument that integrate the various functions provided by the F-TTM module tools and supports operators in taking decisions. In order to embed the functions of the above tools into a unique system the tools are emulated and some decision knowledge is added to some of them. Moreover, new system components are realized, that co-ordinate the needed decision processes. KICKER also emulates the functions provided by the so called *TCM algorithm*, developed under the TCM module of EuROPE-TRIS.

The adoption of a multi-agent approach to the realization of KICKER provides us with the following advantages:

- In KICKER several modules exist, which need to both exchange information and coordinate activities. A model based on a MAS suits perfectly the inherently distributed nature of the application domain and thus allows great scalability and flexibility in the realization of the prototype.
- The decision support given by KICKER will be performed in a highly dynamic environment, under a high degree of information incompleteness and uncertainty. The adoption of the MAS technology helps here in developing entities which perceive the environment and react to the changes occurring in within it, while dealing with incompleteness of their information.
- The agents in a MAS are autonomous, so their behaviour is primarily independent from that of other agents. Thus, modifying the way an agent manages an incoming message can be done by changing a few lines in the agent's code and this change in the agent's behaviour has no impact on the description of the other agents.
- One of the main features of MAS is to provide interfaces toward legacy software, and thus allowing for integration and reuse. This allows us to take advantage of the best features of existing applications, while maintaining the overall description of the system at a high and very realistic level of abstraction.

6.2.1 Background

The starting point of KICKER is the EuROPE-TRIS project, which was funded by the European Commission within the 4th EU-RDT Programme. EuROPE-TRIS has realized three modules that have various purposes.

- The module TCM (Traffic Capacity Management) provides tools that implement procedures for the allocation of railway infrastructures (train path timetables) to railway companies. These procedures are based on EU Directives and rule the transactions between railway companies and track providers. In particular, within the TCM framework the *TCM algorithm* has been developed. Given a timetable, the algorithm is able to determine scheduling conflicts and (by respecting strict input constraints) to supply a new scheduling solution (i.e., a new timetable which does not contain any scheduling conflicts). The TCM algorithm represents the instrument adopted by KICKER in order to solve scheduling conflicts that arise during decisions regarding "New Train" and "Dispatching Authorization" [2].

[2]See Section 6.2.2.2 for a detailed description of decisions supported by KICKER.

- The module TTC (Timetable TeleConference) provides an environment that allows railway operators, working in different sites, to negotiate and validate national or international timetables, without needing to organize more physical meetings.

- The module F-TTM (Freight-Timetable TeleMarketing) supports the activity of contingent train planning, namely the scheduling of trains on a short-term basis. The tools included in this module are described below.

TRAIN: Train Real-time Automatic INformation. This tool helps traffic operators to gather information regarding the status of trains in circulation, in the case where a contingent dispatching decision (e.g., the dynamic scheduling of special or delayed trains among the scheduled ones) must be taken. The tool allows a control centre operator to obtain the status information concerning trains that are presently running on a neighbouring railway network but are going to cross the border and enter the centre's national network.

REMUS: Rail Electronic Mailer Unified System. This tool provides a guide schema for composing multi-lingual structured messages to be exchanged between traffic operators that work for different national railway companies.

VIDES: VIdeo-conferencing Dispatcher Electronic Set. This tool extends the facilities provided by REMUS with multi-media teleconferencing and interactive messaging. VIDES allows for the sharing of documents, in such a way that traffic operators can co-operatively read and possibly modify them.

ITHACA: Inter-modal Train Haulage As Cargo Arrives. This tool provides the infrastructure to link operators at inter-modal centres and yards with traffic co-ordination offices. This infrastructure facilitates the requesting and reservation of new trains, the loading authorization, and the dispatch confirmation for transports. ITHACA creates a direct channel of communication between customers and rail infrastructure management centres where operational decisions have to be taken on a contingent basis.

MIDAS: MIning DAta Server. This tool extends the traditional rolling stock management information systems (legacy architecture) with new open and user friendly interfaces. The tool provides traffic operators with information from a heterogeneous set of data sources. The information is collected and stored into a Rolling Stock Database on a server machine and mainly concerns status of tracks in the stations along a freight railway line and the location of trains that transporting dangerous goods. Traffic operators access the Rolling Stock Database from their client machine, retrieving useful information for

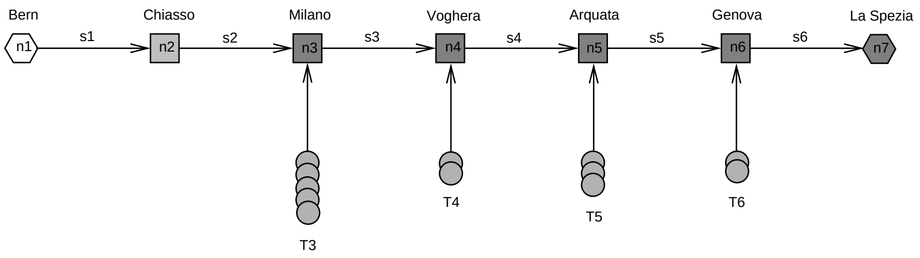

Figure 6.1: Railway line connecting Bern to La Spezia.

the planning and control of fright railway traffic. The operators can also configure a set of alarms to monitor and optimize the allocation of tracks for freight wagons in railway stations.

ODOS: Operational Dispatching On-line Scheduler. This tool provides a graphical display which supports traffic operators in taking decisions regarding new transport requests or the re-dispatching of delayed trains (e.g. verifying the possible allocation of free slots). The tool allows operators of neighbouring countries to schedule new international freight trains that cross national networks at given border stations.

6.2.2 Operative scenario

The operative scenario on which KICKER is supposed to operate is the railway line between Bern (Switzerland) and La Spezia (Italy). This line is particularly interesting because it has been identified by European railways as a freightway, which is heavily utilized by intermodal traffic flows. Such flows originate inside the various container terminals operating along the corridor and terminate in the harbour yards of La Spezia, one of the main Mediterranean ports for container movement.

6.2.2.1 Line configuration

The structure of the line is shown in Figure 6.1. The line is composed of 6 *sections*, each of which runs between a pair of city *nodes*. There is a Swiss section s_1 and 5 Italian sections $s_i, 2 \leq i \leq 6$. There are 7 nodes $n_j, 1 \leq j \leq 7$, that correspond respectively to the cities of Bern, Chiasso, Milan, Voghera, Arquata, Genoa and La Spezia. In the picture, hexagons and squares represent internal and end nodes respectively. Dark grey nodes represent Italian stations, while the light grey node represents the border station between the two countries, and the white node represents the Swiss station of Bern.

Each Italian node (apart form La Spezia) $n_k, 3 \leq k \leq 6$ is connected to a fixed number of *terminals* through minor railway lines. We denote the set of terminals connected to the node n_k with T_k and a terminal belonging to T_k with $t_{sk}, 1 \leq s \leq |T_k|$. Freight trains coming from the inside terminal t_{sk} enter the main line at the node n_k. Chiasso (n_2) is the border node, where freight trains coming from Switzerland enter the Italian network. In our scenario all trains are directed to La Spezia, where the goods they transport are loaded onto ships. For simplicity, we do not consider in KICKER the flow of traffic in the opposite direction (from La Spezia to terminals in Italy or Switzerland).

6.2.2.2 Decisions

In the following, we illustrate the decisions that are supported by KICKER. The decision process for each of them is modelled on the basis of the actual rules which govern the management of freight traffic along the European corridor. At the moment compliance with these rules depends solely on the human traffic operators, who must known how and when each rule needs to be applied.

New Train (NT). This decision is taken whenever a new *unscheduled train* (i.e., a train that has not been scheduled in the *Static Timetable* (ST)) has to be created. Based on constraints posed by the current timetable, this request can be *accepted*, *refused* or *accepted with variations*. In the latter case, the counterproposal can be *accepted* or *refused*.

Loading authorization (LA). Italian freight traffic regulations require terminals to obtain a preliminary authorization for the loading of wagons directed to La Spezia. Based on constraints posed by the foreseen situation in the yard at La Spezia, requests of LA can be *accepted* or *refused*. The loading authorization procedure is required before trains can be dispatched and permits careful management of the destination yard, so as to optimize the flow of wagons toward harbour terminals.

Dispatching authorization (DA). Dispatching authorization is granted depending on the current status of trains in circulation (obtained from the Dynamic Timetable (DT)) and the status of the yards in which the train stops during the journey. A DA is required in two cases:

- whenever a train must access the main railway line from a terminal;
- whenever a train has to enter the Italian railway network via the Chiasso border station.

Dispatching authorization may be either *accepted* or *refused*.

Decisions regarding "new trains" and "loading authorization" are generally made *several hours before* the departure of the freight train. A decision regarding "dispatching authorization" is instead taken *immediately before* the running of the train.

6.2.2.3 Traffic operators

Since KICKER operates on an international basis, we abstract from the organization of the national railway companies and define the generic role of *traffic operator (TO)* to denote all the actors involved in the management of freight train traffic. The role of TO is specialized into the following roles.

Terminal Operator (TerOP). A terminal operator interacts with customers collecting requests for the transportation of goods from the terminal at which she/he works. According to the collected requests, the operator activates the creation of new freight trains departing from that terminal. Just before a train departure date, she/he also requests authorization for the loading of goods onto wagons. Finally, just before the established departure time, she/he requests authorization for the dispatching of the train.

Transport Operator (TrOp). A transport operator organizes the requires personnel and engines, whenever ii is necessary to create of a new train. If the required resources are not available, the request cannot be satisfied. Traffic operators are located in each terminal.

Traffic Co-ordinator (TrCo). The traffic co-ordinator operates in Milan and is responsible for train traffic management along the Italian part of the line from Bern to La Spezia. She/he monitors the overall status of train traffic along the line.

Line Station Operator (LSop). Line station operators work at each of the Italian nodes along the line and are responsible for train traffic management along a particular line section. Their activity is supervised by the Traffic Co-ordinator, who has the authority to override their decisions.

Border Station Operator (BStOp). The border station operator in Chiasso has basically the same responsibilities as any ordinary line station operator. In addition, she/he is required to supply information regarding running trains and the status of yards in the Switzerland railway network.

Production Deputy (PrDe). The Production Deputy works in La Spezia. She/he decides to grant loading authorization for wagons according to the availability of tracks and the presence of dangerous goods in the node of La Spezia.

Table 6.1 lists the roles discussed above, their place of work, and the decisions in which they are involved.

Role	**Working place**	**Decisions**
Terminal Operator (TerOp)	$t_{ij}, 1 \leq i \leq \lvert T_j \rvert, 3 \leq j \leq 6$	NT
Transport Operator (TrOp)	$t_{ij}, 1 \leq i \leq \lvert T_j \rvert, 3 \leq j \leq 6$	NT
Traffic Co-ordinator (TrCo)	n_3 (Milan)	LA, DA
Line Station Operator (LSOp)	$n_i, 3 \leq i \leq 7$	DA
Border Station Operator (BStOp)	n_2 (Chiasso)	DA
Production Deputy (PrDe)	n_7 (La Spezia)	LA

Table 6.1: Role, place of work, and decisions of traffic operators.

6.3 F-TTM tools revised

The F-TTM tools can be grouped into three categories:

Information Sources. These tools provide traffic operators with information about the status of trains in circulation and the yards along the line. Traffic operators make their traffic management decisions based upon this information. Information sources include TRAIN and MIDAS.

Communication Facilities. These tools provide traffic operators with help infrastructure for generic messaging. They are used by traffic operators to exchange information which is not tightly related to train circulation. Communication facilities include REMUS and VIDES.

Decision Facilities. These tools are specifically designed to aid traffic operators during decision processes regarding loading authorization and international dispatching authorization. They provide operators with the ability of automatically exchange relevant information. Decision facilities include ITHACA, ODOS, and TCM.

The F-TTM tools were designed in the EuROPE-TRIS project as tools to help human traffic operators in the activity of contingent train planning. The rationale behind the design of F-TTM tools was to give humans the complete responsibility of decision making and thus the functions provided by F-TTM conform to this expectation. KICKER, on the other hand, is a tool designed to actively suggest possible decisions to human operators who than make the definitive decisions. Towards this aim, we slightly modify some functions provided by F-TTM tools and add decision capabilities to some of them (in particular to Decision Facilities). Moreover, in order to facilitate the realization and evaluation of the first prototype of our decision support system, KICKER simulates the F-TTM tools and the TCM algorithm instead of integrating the real tools themselves.

The simulated F-TTM tools are used in KICKER in the following way.

TRAIN. This tool retrieves information from the Timetable Database regarding either the schedules of trains included in the Static Timetable or the current status of national or international trains, included in the Dynamic Timetable.

REMUS. This tool is used to inform traffic operators of the action suggested by KICKER, once these decisions have been validated by human traffic operators.

VIDES. This tool is activated whenever it is crucial to establish a video conference between traffic operators involved in an international dispatching decision (so that the two operators can reach an agreement).

ITHACA. This module HAS the capability of co-ordinating the process of decision making regarding New Train (NT) and Loading Authorization (LA). In this way, terminal operators and transport operators are supported in the decision process for NT and the traffic co-ordinator and the production deputy are supported in the decision process for LA.

MIDAS. This tool provides information about the yards along the line from Bern to La Spezia, in particular about the presence of dangerous goods.

ODOS. This tool has the capability of co-ordinating decisions regarding Dispatching Authorization (DA). It supports the line station operators, the border station operator, and the traffic co-ordinator in decision making process.

TCM. This tool is used to detect and solve static or dynamic conflicts that can arise when the schedule for a new train is inserted into the timetable or when a train has to be dispatched along the line. TCM takes a (static or dynamic) timetable and a train schedule as input, detects if the given schedule conflicts with schedules included in the timetable, and tries to arrange the train schedule in order to solve conflicts that arose. TCM returns the arranged schedule as output.

6.4 Decision processes

In the following, we describe the decision processes that are used in KICKER to support the decisions that human traffic operators need to make. Decisions refer to requests for NT, LA, and DA. Every decision process involves a set of activities that are performed by different actors (traffic operators and/or revised F-TTM tools) in turn.

Table 6.2 shows the activities that are performed during each decision process, and the actors involved in each activity.

Decision	Activities for decision process	Actors
New Train (NT)	Submission of NT request	TerOp
	Retrieval of human and traction resources	TrOp
	Retrieval of a candidate schedule from ST	TRAIN
	Modification of the candidate schedule	ODOS
	Retrieval of timetable	TRAIN
	Schedule conflict solving	TCM
	Validation of the proposed schedule	TerOp
	Notification of decision to involved TOs	REMUS
Loading authorization (LA)	Submission of LA request	TerOp
	Retrieval of destination yard status	MIDAS
	Decision on LA granting	MIDAS
	Validation of the proposed decision	TerOp
	Notification to involved TOs	REMUS
Dispatching authorization (DA)	Submission of DA request	TerOp, BStOp
	Retrieval of line yard status	MIDAS
	Retrieval of Dynamic Timetable	TRAIN
	Schedule conflict solving	TCM
	Validation of the proposed schedule	TerOp
	Agreement using Video Conference	VIDES
	Notification of decision to involved TOs	REMUS

Table 6.2: Activities for decision processes and involved actors.

6.4.1 New train (NT)

Figure 6.2 depicts the interactions among KICKER actors in the decision process related to the request for a new train.

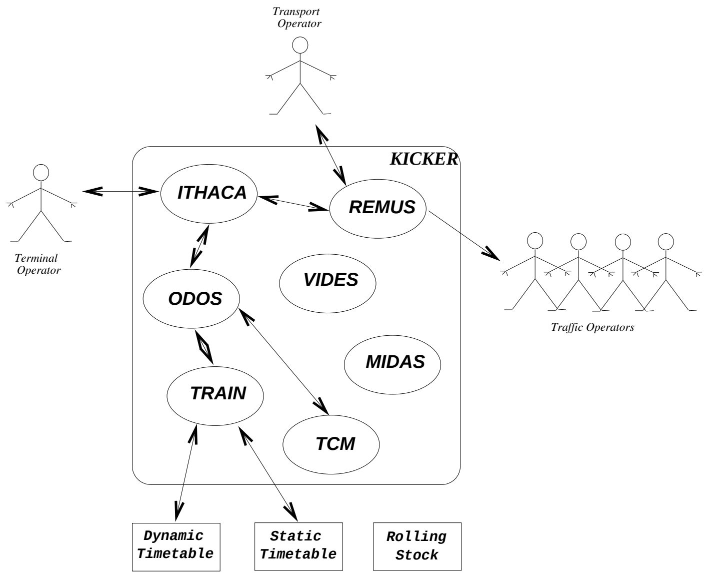

Figure 6.2: Interaction among actors for NT decision.

Submission of NT request. A request for a new train is submitted by a Terminal Operator to ITHACA. It contains the following information:

- departure terminal;
- departure date;
- departure time;
- goods type;
- goods weight.

Retrieval of human and traction resources. Once ITHACA has received the request, it initiates the gathering of human and engine resources. To his aim, ITHACA issues a request, via REMUS, to the Transport Operator that contains the following information:

- goods type;
- goods weight;
- departure date;
- departure time;

The traffic operator selects, on the basis of the weight of the goods, the appropriate engine for the train. The traffic operator then checks whether the engine and personnel for the train will be available at the given departure date and time. The traffic operator answers positively or negatively to ITHACA, via REMUS, according to whether or not all the needed resources will be available.

Retrieval of a candidate schedule from ST. The train schedule is a table containing the train arrival and departure times at all the yards in which it stops during its journey. Once ITHACA has received a positive answer from the Transport Operator, it activates ODOS which, in turn, interacts with TRAIN to retrieve a *candidate schedule* for the train from the Static Timetable. The request from ITHACA to ODOS contains the following information:

- departure terminal;
- departure date;
- departure time.

The request from ODOS to TRAIN contains the same information above.

The Static Timetable contains some programmed schedules for new trains. A candidate schedule is chosen among these schedules in such a way that it fits the submitted request as closely as possible. A candidate schedule contains the following information:

- train number;
- departure terminal;
- departure date;
- candidate departure time;
- schedule of train.

Modification of the candidate schedule. Generally, the departure time proposed by a candidate schedule does not correspond to the requested one. If this is the case, ODOS operates a time shifting of the candidate departure time, to make it conformant with the requested one. Consequently, all the times contained in the candidate schedule are coherently modified.

Retrieval of timetable. After ODOS has modified the candidate schedule it needs to activate TCM in order to detect and solve schedule conflicts. First, ODOS queries TRAIN. Queries contain the following information:

- train number;
- departure terminal;
- departure date;
- modified departure time.

Queries can be processed by TRAIN in two different ways:

- if the modified departure time is more than six hours ahead the current time[3], TRAIN queries the Static Timetable;
- otherwise, TRAIN queries the Dynamic Timetable.

TRAIN returns to ODOS a partial timetable, which TCM takes as input along with the modified candidate schedule. TCM detects possible schedule conflicts and arranges the modified candidate schedule in order to eliminate these conflicts. The *proposed schedule* is returned to ODOS.

Validation of the proposed schedule. ODOS returns the proposed schedule to the TerOp via ITHACA. TerOp may either validate the proposed schedule, or submit a counterproposal to KICKER, in order that it be evaluated and possibly modified by the system. If this is the case, a new proposal will be returned once more to TerOp, which can either accept or refuse it.

Notification of decision to involved TOs. Once a proposed schedule has been validated, REMUS is activated to notify all the involved traffic operators. Messages sent to Traffic Operators contain the following information:

- train number;
- departure terminal;
- departure date;

[3]Six hours is the maximum estimated time that a train can take to reach La Spezia from Bern.

- departure time;
- goods type;
- goods weight.

6.4.2 Loading authorization (LA)

Figure 6.3 depicts the interactions among KICKER actors in the decision process related to the request for loading authorization.

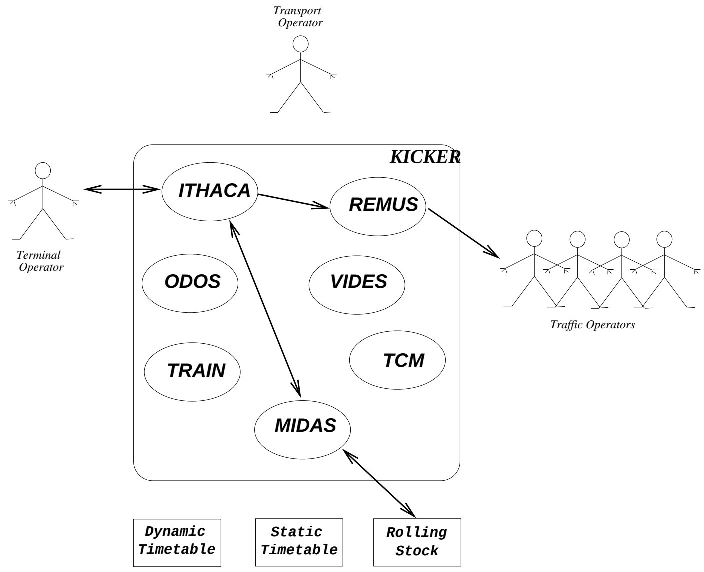

Figure 6.3: Interaction among actors for LA decision.

Submission of LA request. Terminal Operators submit requests for loading authorization to ITHACA. Requests contain the following information:

- train number;
- departure terminal;
- arrival date;
- number of wagons;
- amount of dangerous goods.

Retrieval of destination yard status. ITHACA activates MIDAS which retrieves information regarding the expected status of the destination yard (i.e., La Spezia) on arrival date[4] from the Rolling Stock module.

Responses from the Rolling Stock module contain the following information:

- occupancy index for the destination yard on arrival date;
- dangerous goods stocking index for the destination yard on arrival date.

The occupancy index gives a measure of how many available tracks there will be in the destination yard to receive trains. The dangerous goods stocking index measures the amount of dangerous goods in the destination yard.

Decision on LA granting. MIDAS knows the maximum allowed occupancy index for the destination yard, as well the maximum amount of allowed dangerous goods. According to the information received from the Rolling Stock module MIDAS can decide whether or not to grant the loading authorization.

Validation of the proposed decision. The reply to the request for LA is proposed to the Traffic Operator via ITHACA. The traffic operator can validate or override the proposed decision.

Notification to involved TOs. If the loading authorization is validated, REMUS is activated to notify all the involved traffic operators.

Dispatching authorization (DA)

Figure 6.4 depicts the interactions among KICKER actors in the decision process related to the request for dispatching authorization.

Submission of DA request. A request for dispatching authorization is submitted by a Terminal Operator or by the Border Station Operator to ODOS. The request contains the following information:

- train number;

[4]It is supposed that a train can reach its destination by the arrival date, but the exact time of arrival is not known.

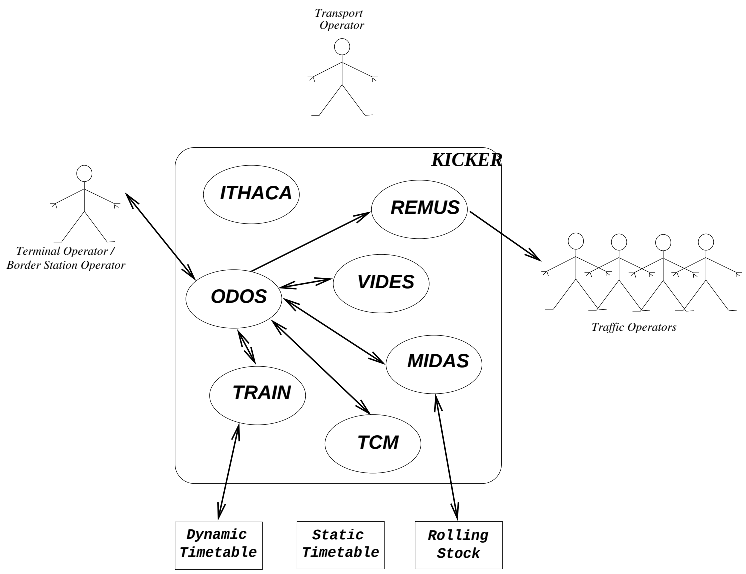

Figure 6.4: Interaction among actors for DA decision.

- departure place;
- schedule of train;
- amount of dangerous goods.

Departure place may be either a terminal or the border node of Chiasso. The schedule of the running train is a table containing arrival and departure times in all the yards in which the train stops.

Retrieval of line yard status. Once a request has been submitted, ODOS interacts with MIDAS. MIDAS collects from the Rolling Stock module information concerning the status of dangerous goods in every yard in which the train will stop. MIDAS knows what the maximum allowable amount of dangerous goods is for each yard. If there is any yard at which the arrival of the train would cause the maximum amount to be exceeded, then the dispatching authorization for the train cannot be granted.

Retrieval of Dynamic Timetable If ODOS receives a positive answer from MIDAS, it asks TRAIN to query the Dynamic Timetable, which returns a partial dynamic timetable based on the train's departure time. Queries contain the following information:

- train number;

- departure terminal;
- departure date.

Schedule conflict solving ODOS passes the train schedule and the partial dynamic timetable (just received from TRAIN) to TCM, which performs schedule conflict resolution, as previously described.

Validation of the proposed schedule. The terminal operator or the border station operator validates the proposed schedule.

Agreement via Video Conference. VIDES may be activated in the case where the Border Station Operator did not validate the proposed dispatching schedule. In this case, it could be the case that the Border Station insists on the departure of the train and makes the necessary agreements with the Traffic Co-ordinator in Milan.

Notification of a decision to involved TOs. If the dispatching authorization is granted, REMUS is activated to notify all the involved traffic operators.

6.5 Prototyping KICKER in CaseLP

This section describes how KICKER has been prototyped using the CaseLP method and tools.

6.5.1 Step 1: informal description of the features of the application

The functions that KICKER must provide to traffic operators were described in Section 6.4. The decisions that are supported by the system, as well as the functions provided by the emulated F-TTM modules (see Section 6.3), represent the tasks that KICKER performs and give the task organization for the prototype. The description of the railway line given is Section 6.2.2.1, and the information exchanged among actors during decision processes (see Section 6.4) gives the knowledge organization for the prototype.

Exported services: $S^e = \{NTP, LAP, DAP, NTN, LAN, DAN\}$
NTP: New Train Proposal
LAP: Loading Authorization Proposal
DAP: Dispatching Authorization Proposal
NTN: New Train Notification
LAN: Loading Authorization Notification
DAN: Dispatching Authorization Notification

Imported services: $S^{imp} = \{HTRA, NTC, CSR, TR, DTR, DYS, LYS\}$
HTRA: Human and Traction Resources Availability
NTCP: New Train Counterproposal
CSR: Candidate Schedule Retrieval
TR: Timetable Retrieval
DTR: Dynamic Timetable Retrieval
DYS: Destination Yard Status
LYS: Line Yards Status

Internal services: $S^{int} = \{HTRA, PSR, CSR, TR, SCS, LAG, DAPG, DTR, VC\}$
HTRA: Human and Traction Resources Availability
PSR: Proposed Schedule Retrieval
CSR: Candidate Schedule Retrieval
TR: Timetable Retrieval
SCS: Schedule Conflict Solving
LAG: Loading Authorization Grant
DAPG: Dispatching Authorization Partial Grant
DTR: Dynamic Timetable Retrieval
VC: Video Conference

Figure 6.5: Service analysis.

6.5.2 Step 2: service analysis

Based on the output of Step 1, the services that the prototype provides are formally defined. Figure 6.5 shows the set of services that have been identified in the prototyping of KICKER. The prototype exports to human users the set of services S^e. In this set, services NTP, LAP, and DAP correspond to the three main decisions that the system supports. Services NTN, LAN, and DAN are exported to traffic operators, in order to notify them about the decisions that the system takes.

The set S^{imp} includes the services that are imported into KICKER, and are provided by human users and external software modules. In particular, $HTRA$ and $NTCP$ are provided from human transport and terminal operators respectively, and refer to the availability of resources for a new train and counter proposals for new train schedules. Services CSR, TR, and DTR are imported from the Timetable Database to retrieve a candidate schedule for a new train, and static and dynamic timetables used for solving possible schedule conflicts. The timetables (both static or dynamic) are used during the decision process regarding a new train. The dynamic timetable is used during the decision process regarding dispatching authorization. Services DYS and LYS are imported from the Rolling Stock Database and provide information regarding the status of the destination yard (used for granting a loading authorization) and the status of the stations along the railway line (used for granting a dispatching authorization).

Internal services S^{int} are provided and requested inside the prototype. It is straightforward process to find a correspondence between these services and the activities that compose decision processes (see table 6.2).

6.5.3 Step 3: ontological analysis

The description related to knowledge organization given in Step 1 guides the definition of a domain ontology for this problem. For example, the domain ontology includes representation for the topological structure of the line. Figure 6.6 shows part of the domain ontology related to the topological structure of the railway line.

Further than defining the domain ontology, this step also defines the domain dependent part of the ontology *agent* (see section 4.3.2). Toward this aim, the ontology *agent* includes the predicate $domain\_service/2$ used to define domain level services. A term s and a predicate $domain\_service(s)$ are added to *agent* for each domain service s selected in Step 2. The knowledge base for internal services and for services exported or imported to/from humans is shown in Figure 6.7.

6.5.4 Step 4: prototype architectural description

In this step, MAS-adl is used to define the static structure of the MAS which realizes the prototype of KICKER. Figures 6.8 and 6.9 show the definition of agent templates and instances, and the agent communication structure which is defined by links between services. The natural choice here is to define seven agent templates, one for each of the revised F-TTM tool. All these templates are given a reactive architecture, The two interface agent templates (`Train` and `Midas`) allow access to external modules, using the interpreters provided. Each template exports, imports, provides, or requires a subset of the services previously defined, in order to properly perform its task in the decision making process.

The MAS that realizes KICKER is composed of exactly one instance of each template. Actually, in order to execute the prototype in CaseLP, we also define four "dummy" agents

Ontology name: *kicker*

Terms: $t_1, \ldots, t_n, bern, chiasso, milano, voghera, arquata, genova, laspezia$

Predicates:

$terminal/1, node/1, initial\_node/1, end\_node/1, border\_node/1, italian\_node/1,$
$precede/2, before/2, linked\_to/2$

Knowledge base:

$terminal(t_1), \ldots, terminal(t_n),$
$node(bern), \ldots, node(laspezia), initial\_node(bern), end\_node(laspezia), border\_node(chiasso),$
$italian\_node(milano), \ldots, italian\_node(laspezia)$
$precede(bern, chiasso), \ldots, precede(genova, laspezia)$
$before(X, Y) \leftarrow precede(X, Y)$
$before(X, Y) \leftarrow precede(X, Z) \wedge before(Z, Y)$
$linked\_to(t_{m1}, milano), \ldots linked\_to(t_{ms}, milano)$
$\ldots$
$linked\_to(t_{g1}, genova), \ldots linked\_to(t_{gt}, genova)$

Constraints:

$precede(X, Y) \rightarrow node(X) \wedge node(Y)$
$linked\_to(T, N) \rightarrow terminal(T) \wedge italian\_node(N)$

Figure 6.6: Ontological analysis (1).

which acts on behalf of human operators. These agents are called *ter_op*, *bordsat_op*, *transp_op*, and *traff_op*. They act on behalf of terminal, border station, transport, and traffic operators respectively.

The required and provided services are linked using communication links. Figure 6.9 depicts links for internal and external services. Link specification also involves "dummy" agents.

6.5.5 Step 5: conversation specification

This step strictly relates to *interaction organization* and concerns the definition of *conversations* that are used to provide services in the prototype.

First, message contents and their association to services are included in the domain dependent part of the *agent* ontology. Figure 6.10 shows the ontology, limited to certain messages and services regarding decisions about new trains.

The predicate *domain_msg* holds for all the message contents that are defined. The predicate *domain_associated_to* holds for pairs of the form $\langle message\_content, service\rangle$. The content of the messages has been chosen in agreement with the structure of the information exchanged among the actors (see Section 6.4).

New Train:
domain_service(*ntp*), *domain_service*(*htra*), *domain_service*(*psr*), *domain_service*(*csr*), *domain_service*(*tr*), *domain_service*(*scs*), *domain_service*(*ntcp*), *domain_service*(*ntn*).

Loading authorization:
domain_service(*lap*), *domain_service*(*lag*), *domain_service*(*lan*).

Dispatching authorization:
domain_service(*dap*), *domain_service*(*dapg*), *domain_service*(*dtr*), *domain_service*(*scs*), *domain_service*(*vc*), *domain_service*(*dan*).

Figure 6.7: Ontological analysis (2).

In Figure 6.10 the first, second, and third message contents are associated to the service *ntp*. The first content is used to ask *ithaca* about a proposal for a new train schedule. The second message content is sent by *ithaca* in reply to a proposal request. The variable *Proposal* is instantiated by *ithaca* with a proposed schedule. The third message content is used to tell *ithaca* whether or not a schedule proposal has been accepted by a terminal operator.

An internal or external conversation is set for each link defined in Step 4. Some conversations related to decision about new trains are shown as an example in Figure 6.11. The external conversation c_1 is related to the service NTP, which is exported by KICKER to terminal operators. The internal conversation c_2 is related to the service $HTRA$, which is provided by agent *remus* and requested by agent *ithaca*. The external conversation c_3 involves *remus* and *transp_op*, and is related to the service $HTRA$, which is imported by KICKER from the external human transport operator. The internal conversation c_4 is started by agent *ithaca* to ask *odos* about a proposed schedule for a new train.

Sub-conversations are then defined, an example is given below.

$$c_1 sc_{m_1} c_2, \quad c_2 sc_{m_1} c_3, \quad c_2 sc?_{m_2} c_4$$

In the example, conversation c_2 starts when message m_1 of conversation c_1 has been received by *ithaca*. Analogously, c_3 starts when *remus* receives the first message of c_2. Conversation c_4 can possibly start after the conclusion of c_2, depending whether or not resources for a new train are available or not.

6.5.6 Step 6: initial state and program description

This step uses AgentRules to set the initial state and program of the agents in the prototype. The initial state of each agent instance is given as a set of facts, which represents its initial

```
agenttemplate Ithaca {
   kind: logical;
   architecture: reactive;
   exports: NTP, LAP;
   imports: NTCP;
   provides: NTN, LAN;
   requires: HTRA, PSR, LAG;
}

agenttemplate Train {
   kind: interface;
   architecture: reactive;
   interpreter: timetable_int;
   imports: CSR, TR, DTR;
   provides: CSR, TR, DTR;
}

agenttemplate Vides {
   kind: logical;
   architecture: reactive;
   provides: VC;
}

agenttemplate Remus {
   kind: logical;
   architecture: reactive;
   imports: HTRA;
   exports: NTN, LAN, DAN;
   provides: HTRA;
   requires: NTN, LAN, DAN;
}

agenttemplate Odos {
   kind: logical,
   architecture: reactive;
   exports: DAP,
   provides: PSR, DAN;
   requires: CSR, TR, DAPG, DTR,
             SCS,VC;
}

agenttemplate Midas {
   kind: interface;
   architecture: reactive;
   interpreter: rolling_stock_int;
   imports: DYS, LYS;
   provides: LAG, DAPG;
}

agenttemplate Tcm {
   kind: logical,
   architecture: reactive,
   provides: SCS;
}

agentInstances {
   1 ithaca Ithaca;
   1 remus Remus;
   1 vides Vides;
   1 odos Odos;
   1 train Train;
   1 tcm Tcm;
   1 midas Midas;
}
```

Figure 6.8: Prototype architectural description (1).

knowledge about the application domain. Specification of agent programs is based on the conversation model defined in Step 5. Figure 6.12 and 6.13 show some of the rules that are included in the programs of agents *ithaca* and *ter_op* to execute the conversation c_1. In figure 6.12, the first rule states that whenever agent *ithaca* receives a request for a schedule proposal (both state and auxiliary conditions are *true*), it records the request in its state (*assert_state*) and submits a request for human and traction resources to *remus* which forwards it to *transp_op*. The second rules states that when agent *ithaca* receives from *ter_op* a message that validates a proposed schedule, it queries its internal state to retrieve the details regarding the corresponding request for new train (previously received) and then activates *remus* in order to notify the traffic operators of the decision.

In Figure 6.13, the first rule is used by the agent to submit a counterproposal for a new train schedule to *ithaca*. The second rule submits to *ithaca* an acceptance message for a

```
link {
   ter_op.NTP <- ithaca.NTP,          ithaca.HTRA <- remus.HTRA,
   remus.HTRA <- transp_op.HTRA,      ithaca.PSR <- odos.PSR,
   odos.CSR <- train.CSR,             odos.TR <- train.TR,
   odos.SCS <- tcm.SCS,               ithaca.NTCP <- ter_op.NTCP,
   remus.NTN <- ithaca.NTN,           traff_op.NTN <- remus.NTN,

   ter_op.LAP <- ithaca.LAP,          ithaca.LAG <- midas.LAG,
   remus.LAN <- ithaca.LAN,           traff_op.LAN <- remus.LAN,

   ter_op.DAP, bord_stat_op.DAP <- odos.DAP,
   odos.DAPG <- midas.DAPG,           odos.DTR <- train.DTR,
   odos.VC <- vides.VC,               remus.DAN <- odos.DAN,
   traff_op.DAN <- remus.DAN
}
```

Figure 6.9: Prototype architectural description (2).

proposed schedule. A third rule (not shown) is used by *ter_op* to handle a message stating that there are no available resources for the new train.

6.5.7 Step 7: prototype implementation

In this step, the programs for the agents defined in Step 6 have been translated into ProlAg. The interpreters needed to interface with external software modules (Timetable and Rolling Stock Databases), have not actually been implemented. Instead, these modules have been simulated into the code of the agents *train* and *midas*.

6.5.8 Step 8: prototype execution, testing, and simulation

This step had been performed using the CaseLP Visualizer. On line visualization helped very much in the discovery of implementation errors and of missing rules in the agent programs. Off line visualization was instead used to provide an overall trace of the behaviour of KICKER. In this way, the prototype could be validated by experts of Italian Railways.

Figure 6.14 presents an example of execution tracing. The trace records the messages exchanged by KICKER agents during the simulated decision process for managing a new train. The figure only shows some of the exchanged messages (limited to the conversations $c_1, \ldots c_4$). Note (in the upper left corner) that agent *ithaca* asks *odos* about a proposed schedule for the new train, after it has received a positive answer from *transp_op* (not shown) regarding the availability of human and traction resources for the train.

Figure 6.15 presents the complete trace of messages for a request for loading authorization.

$domain\_msg(ntp(Terminal, Date, Time, GType, GWeight, Proposal))$
$domain\_associated\_to(ntp(Terminal, Date, Time, GType, GWeight, Proposal), ntp)$
$domain\_msg(ntp(Terminal, Date, Time, Proposal))$
$domain\_associated\_to(ntp(Terminal, Date, Time, Proposal), ntp)$
$domain\_msg(ntp(Terminal, Date, Time, Proposal, Accepted))$
$domain\_associated\_to(ntp(Terminal, Date, Time, Proposal, Accepted), ntp)$
$domain\_msg(htra(GType, GWeight, Date, Time, Answer))$
$domain\_associated\_to(htra(GType, GWeight, Date, Time, Answer), htra)$
$domain\_msg(psr(Terminal, Date, Time, Proposal))$
$domain\_associated\_to(psr(Terminal, Date, Time, Proposal), psr)$

Figure 6.10: Ontological analysis (3).

In this simulation KICKER proposes a solution that is validated by *ter_op*. Then, it is notified (via *remus*) to *traff_op* (not shown).

6.6 Discussion

From the point of view of FS-S.p.A., the main goal of the project was to verify the feasibility of an automatic decision support system for the complex scenario of railway freight transport. From the point of view of DISI, the main goal of the KICKER project was to test CaseLP against a real world, not trivial case study, and to obtain useful feedback for its improvement. The implemented prototype, even if realized adopting many simplifications with respect to the real scenario, achieved both the goals above. This section outlines the encouraging results that have been obtained.

6.6.1 Results for FS-S.p.A.

Italian railways are quickly moving towards wide area train control systems, equipped with a high degree of "intelligence", for the management of conflicts generated by the running of trains and the maintenance of the safety requirements. Decision support systems need offer a global framework for capturing the wide set of processes involved in train dispatching (and for modelling their relationships, priority and inherent procedures), in order that train operators can be provided with more powerful decision support. Italian Railways are naturally interested in demonstrating the feasibility of different technologies for modelling the core railway management processes.

Conversation c_1 (ter_op.NTP $\leftarrow$ ithaca.NTP)
Message m_1*:* ter_op $\rightarrow$ ithaca
Performative: ask *Content:* ntp(Terminal, Date, Time, GType, GWeight, Proposal)
Message m_2*:* ithaca $\rightarrow$ ter_op
Performative: reply *Content:* ntp(Terminal, Date, Time, Proposal)
Message m_3*:* ter_op $\rightarrow$ ithaca
Performative: tell *Content:* ntp(Terminal, Date, Time, Proposal, Accepted)
Conversation c_2 (ithaca.HTRA $\leftarrow$ remus.HTRA)
Message m_1*:* ithaca $\rightarrow$ remus
Performative: ask *Content:* htra(GType, GWeight, Date, Time, Answer)
Message m_2*:* remus $\rightarrow$ ithaca
Performative: reply *Content:* htra(GType, GWeight, Date, Time, Answer)
Conversation c_3 (remus.HTRA $\leftarrow$ transp_op.HTRA)
Message m_1*:* remus $\rightarrow$ transp_op
Performative: ask *Content:* htra(GType, GWeight, Date, Time, Answer)
Message m_2*:* transp_op $\rightarrow$ remus
Performative: reply *Content:* htra(GType, GWeight, Date, Time, Answer)
Conversation c_4 (ithaca.PSR $\leftarrow$ odos.PSR)
Message m_1*:* ithaca $\rightarrow$ odos
Performative: ask *Content:* psr(Terminal, Date, Time, Proposal)
Message m_2*:* odos $\rightarrow$ ithaca
Performative: reply *Content:* psr(Terminal, Date, Time, Proposal)

Figure 6.11: Conversation specification.

The MAS approach adopted in KICKER seems to be particularly suitable to formalize such processes and thus represents a valid framework for the design of a decision support system for train operators. CaseLP has demonstrated to be a powerful instrument for the rapid prototyping of MAS applications. The availability of such a tool increases the likelihood of MAS technology adoption by Italian Railways, because the modelled system can be quickly and extensively simulated.

The MAS approach is also of methodological importance, due to the current lack of use of formal methods specification and development of railway systems. In fact, KICKER represents an example of how a rigorous approach to application design and realization can produce positive results, in terms of the exact and quick validation of system specifications, and the early testing of the system features and performance (by means of a simulation platform).

```
on_message
   ask {
      content: ntp(Terminal, Date, Time, GType, GWeight, Proposal),
      sender: ter_op,
      receiver: ithaca
   }
check
   true and true
do
   assert_state(request(ntp(Terminal, Date, Time, GType, GWeight))),
   send(
      ask {
         content: htra(GType, GWeight, Date, Time, Answer),
         sender: ithaca,
         receiver: remus
      },
      remus
   )

on_message
   tell {
      content: ntp(Terminal,Date,Time,schedule(TrainNo,_,_,Schedule),yes),
      sender: ter_op,
      receiver: ithaca
   }
check
   request(ntp(Terminal, Date, Time, GType, GWeight)) and true
do
   send(
      tell {
         content: ntn(TrainNo, Terminal, Date, Time, GType, GWeight, Schedule),
         sender: ithaca,
         receiver: remus
      },
      remus
   )
```

Figure 6.12: Program description for agent *ithaca*.

6.6.2 Results for DISI

We can distinguish three main results that have been obtained in this project.

Applicability of CaseLP. The realization of KICKER is a further demonstration that CaseLP is profitably applicable for the prototyping and simulation of transport applications, as already shown in [55] and [3]. CaseLP suits very well the realization of prototypes in which several entities have to co-operate and co-ordinate in order to achieve a common goal and must massively use message passing in order to exchange information.

CaseLP method. The development method included in CaseLP focuses mainly on the *architectural design* and *implementation* of prototypes, and only partially on *requirement*

```
on_message
   reply {
      content: ntp(Terminal, Date, Time, Proposal),
      sender: ithaca,
      receiver: ter_op
   }
check
   nt_counterproposal(Terminal, Date, Time, yes) and
   modify(Proposal, Cproposal)
do
   send(
      ask {
         content: ntcp(Terminal, Date, Time, CProposal),
         sender: ter_op,
         receiver: ithaca
      },
      ithaca
)

on_message
   reply {
      content: ntp(Terminal, Date, Time, Proposal),
      sender: ithaca,
      receiver: ter_op
   }
check
   nt_acc(Terminal, Date, Time, yes)
do
   send(
      tell {
         content: ntp(Terminal, Date, Time, Proposal, yes),
         sender: ter_op,
         receiver: ithaca
      },
      ithaca
   )
```

Figure 6.13: Program description for "dummy" agent *ter_op*.

analysis. The development of a complex application such as KICKER has highlighted the need to introduce into our method more abstract models to capture also the "early" and "late" requirements for the application. In fact, we capture requirements only informally, in Step 1 of our method. The adoption of a more formal phase for requirements analysis (for example, as Tropos does (see Section 2.5.4)) could prevent a too early commitment to a sub-optimal MAS architecture (especially in terms of the incorrect grouping of functions into agents) for the application under development.

CaseLP tools. KICKER has been a very good testbed for the set of tools and notations that CaseLP provides. MAS-adl, the language for architectural description of prototypes, has demonstrated to be a useful tool for setting up the configuration of the prototype (particularly in describing the links that agents use for service communication). The specification of conversations in KICKER has been performed quite naturally in CaseLP. The conversation specification has been very useful for defining the agent programs (thanks to the quite direct mapping between messages that compose conversations and program rules rules that handle them). The CaseLP Visualizer proved its usefulness when debugging the agent code. In particular, the on line visualization of prototype execution allowed for the discovery of errors in the agent rules, and highlighted the need to introduce rules which were missing. The off line visualization has been used to present the KICKER decision making processes to experts from Italian Railways. Off line traces give a comprehensive overview of the decision processes and have been used to validate and optimize them.

On the other hand, many tools should be improved or added to CaseLP. For example, translation between AgentRules and ProlAg need to be fully automated. The realization of such a compiler would not require a great effort. Moreover, AgentRules code for some very common communication patterns should be provided by the system, in order to speed up prototype realization. In general, the CaseLP tools seem quite adequate for modelling communicative agents, while they seem to be a little unsatisfactory for modelling agents with knowledge and reasoning. The communicative nature of the prototypes that have been realized using CaseLP has certainly influenced the incremental design of our system. The adoption of more sophisticated knowledge representation and reasoning techniques is now necessary to allow for the use of CaseLP in a wider range of application areas.

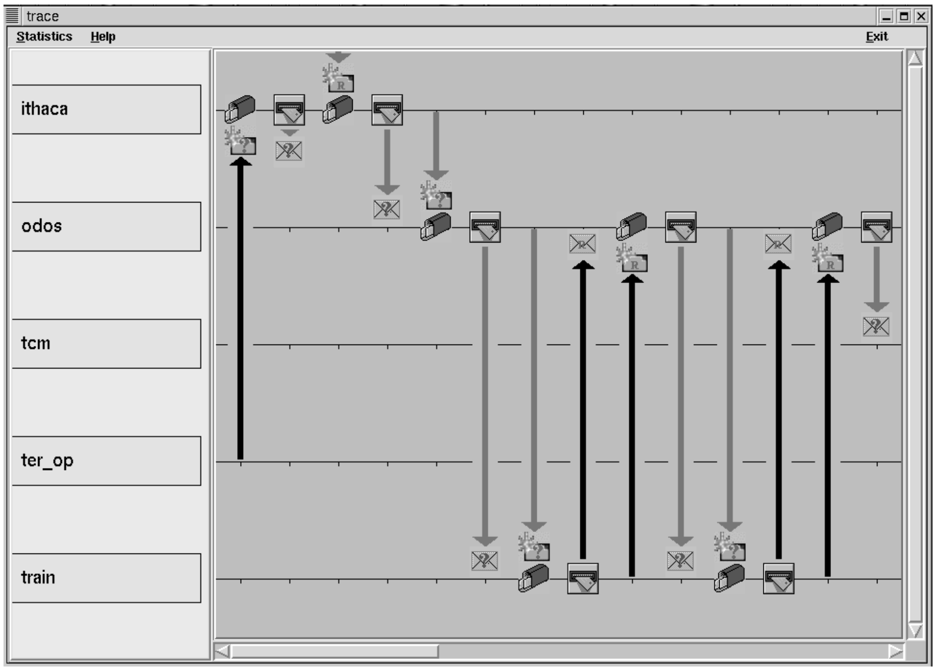

Figure 6.14: Tracing of NT decision process.

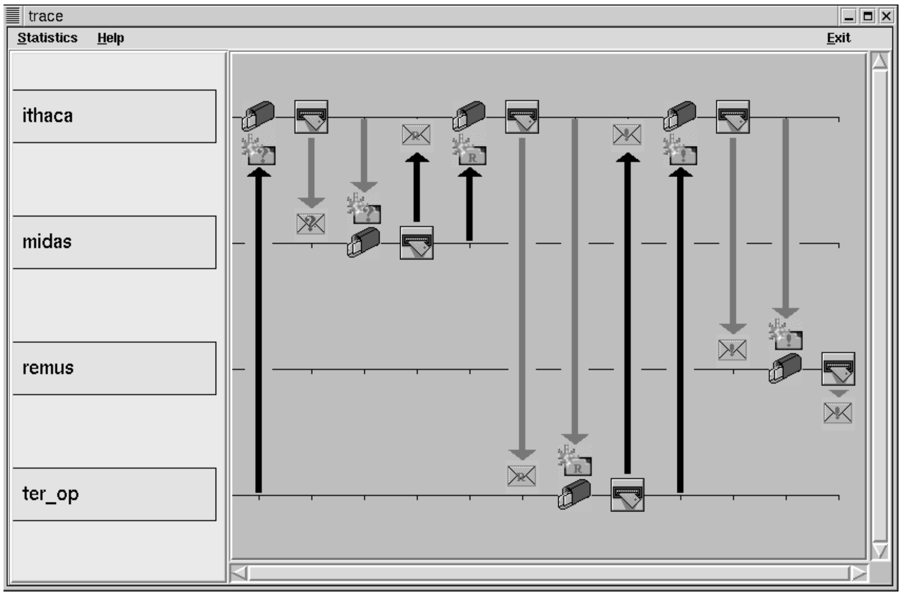

Figure 6.15: Tracing of LA decision process.

Chapter 7

Conclusions and future developments

This dissertation has presented CaseLP (*Complex Applications Specification Environment based on Logic Programming*), an experimental logic-based prototyping environment for building complex and distributed software applications using as multi-agent systems. The basic idea behind the design of CaseLP is to create an environment which can accommodate various specification and implementation languages as well as legacy software, so that a MAS prototype can be built using a range of techniques integrated into a common framework based upon logic programming. In this chapter we summarize the main results that have been achieved so far, and give an overview of the future research directions in which CaseLP is evolving.

7.1 Achieved results

The research on CaseLP has been undertaken with two main objectives in mind. Our first goal was showing that logic programming could facilitate the realization of an environment for the (semi) formal development of agent-based prototypes. The second goal was to demonstrate that such an environment could be effectively used for rapid prototyping of complex real-world software applications.

As far the first goal is concerned, we consider the current status of our environment to be satisfactory. The CaseLP method allows a software engineer to develop a prototype MAS following a sequence of clear steps and CaseLP tools support the developer in performing the various steps. The CaseLP method and tools address three orthogonal issues that are fundamental in the development of multi-agent systems. *Task organization* helps the designer to consider the functional structure of the MAS, to identify the basic services that that prototype has to provide, and to group them into agents. *Task organization* is described using the language MAS-adl, which specifies the architecture of the prototype in terms of agent templates, instances, and communication links. *Knowledge organization* concerns the declarative or procedural knowledge that agents use and manipulate in order to accomplish their tasks. Declarative knowledge is defined in CaseLP by means of ontologies, that allow the designer to express knowledge about the application domain of the MAS prototype, and to formalize the attributes and functionalities of CaseLP agents. Procedural knowledge concerns the rules and plans the agents use in order to perform the tasks that they have been assigned. The language AgentRules allows the designer to program the behaviour of the agents in terms of of easy-to-use, high-level rules. *Interaction organization* concerns the way agents interact with each other. CaseLP addresses this issue by providing the software engineer with a sophisticated agent communication language (KQML) and with tools for the specification of conversations that take place between agents. This allows the designer to define high-level, message-oriented conversations and protocols for information exchange. CaseLP also supports the execution, simulation, and testing of MAS prototypes. The CaseLP Visualizer provides a graphical environment that allows for interactive debugging of the prototype implementation. The visualizer allows the designer to stop the execution of the MAS at any point and observe in details the exchanged messages and the agent state updated. Moreover, the visualizer provides an off-line tracing tool that can be used to present the "big picture" of interactions among agents. The CaseLP Shell allows the user to set parameters for the simulation (such as the duration of the simulation, or the failure rate of communication channels between agents) and collect statistics from simulation execution.

As far as the second goal is concerned, CaseLP has so far been successfully applied to the realization of several prototypes. As well as the two applications outlined is Section 3.5, the fruitful co-operation with Elsag S.p.A. has also led to the realization of a prototype for a system that simulates a postal device performing mail sorting [18]. Moreover, two other prototype applications were built to retrieve medical information contained in distributed databases [70] and to manage distributed database transactions [59], where agent-oriented and constraint logic programming techniques were combined. This dissertation has presented in detail KICKER, a joint project between the Italian Railways (FS-S.p.A.) and our department. We believe that this prototype is a very good demonstration of the applicability of CaseLP to complex real-word applications.

7.2 Future work

CaseLP is an ongoing research project and there are many directions for future work. Here, we outline the most important ones [58].

Distributed CaseLP. Currently, the agents that build up a CaseLP prototype are only logically distributed and they are executed by a unique, centralized simulator. One research direction aims to physically distribute the agents and the simulation. A unique centralized entity will need to intercept significant events occurring at the distributed nodes in order to trace the global execution of the simulation and to provide information on it. Nevertheless, the agents will be able to run autonomously and asynchronously at the different nodes of the network, and thus the prototype will gain in efficiency and similarity to the final, distributed application.

Multi-language agent integration. Another research issues involves extending the set of languages that can be used to specify agents within CaseLP. We aim to allow prototype developers to write agents programs not only in AgentRules (or directly in ProlAg), or specify agent architectures using $\mathcal{E}_{hhf}$ or HEMASL, but also integrate into the prototype agents written in more widespread and commercial languages, such as Java, or C++. Doing this would greatly help with the problem of legacy software integration, since agents written in these languages can easily be used as wrappers for legacy software.

Integration in CaseLP of new specification languages. Languages and notations provided by CaseLP for the specification of agents and MAS capture only some of their characteristics, but no language describes all their features. One possibility is to develop a new specification language for multi-agent systems which is able to capture and represent both the distribution of the agents and the integration of legacy software. This language should allow the description of simple reactive agents as well as complex agents characterized by mentalic features. A well founded semantics for the language should also be given, in order to help bridge the gap between theory and practice in the agent field.

Another possibility is to adopt some existing specification languages for use in the description of MAS. In [22], an extensive analysis of UML [28] as multi-agent specification language is presented. This work also extends UML with some agent-oriented specification constructs and gives an approach for the integration of a textual version of the extended UML into CaseLP.

Bibliography

[1] A. Abderrahamane, D. Chan, P. Dufresne, E. Falvey, H. Grant, A. Herold, G. Macartney, M. Meier, D. Miller, S. Mudambi, S. Novello, B. Perez, E. van Rossum, J. Schimpf, K. Shen, P. A. Tsahageas, and D. H. de Villeneuve. *ECLiPSe 4.2 User Manual.* IC-Parc, London, 1999.

[2] S. Adali and V. S. Subrahmanian. Intelligent Caching in Hybrid Knowledge Bases. In N. Mars, editor, *Proc. of 1995 International Conference on Very Large Knowledge Bases*, pages 247–256, Twente, The Netherlands, May 1995. IOS Press.

[3] E. Appiani, M. Martelli, and M. Mascardi. A Multi-Agent Approach to Vehicle Monitoring in Motorway. Technical Report DISI-TR-00-13, DISI – Università di Genova, Genova, Italy, 2000.

[4] A. Aretti. Semantica di Sistemi Multi-Agente in Logica Lineare. Master's thesis, DISI – Università di Genova, Genova, Italy, 1999. In Italian.

[5] J. Bowen. The World Wide Web Virtual Library: Formal Methods, 2000. http://archive.comlab.ox.ac.uk/formal-methods.html.

[6] M. Bozzano, G. Delzanno, M. Martelli, M. Mascardi, and F. Zini. Logic Programming & Multi-Agent Systems: a Synergic Combination for Applications and Semantics. In *The Logic Programming Paradigm: a 25-Year Perspective.* Springer-Verlag, 1999.

[7] M. Bozzano, G. Delzanno, M. Martelli, M. Mascardi, and F. Zini. Multi-Agent Systems Development as a Software Engineering Enterprise. In G. Gupta, editor, *Proc. of First International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, number 1551 in Lecture Notes in Computer Science, San Antonio, Texas, January 1999. Springer-Verlag.

[8] F. M. T. Brazier, B. Dunin Keplicz, N. R. Jennings, and J. Treur. DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework. *International Journal of Cooperative Information Systems*, 6:67–94, 1997. Special Issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems.

[9] A. Brink, S. Marcus, and V. S. Subrahmanian. Heterogeneous Multimedia Reasoning. *IEEE Computer*, 28(9):33–39, September 1995.

[10] R. A. Brooks. Intelligence without Representation. *Artificial Intelligence*, 47:139–159, 1991.

[11] B. Burmeister, A. Haddadi, and G. Matylis. Application of Multi-Agent Systems in Traffic and Transportation. *IEE Proc. of Software Engineering*, 144(1):51–60, 1997.

[12] P. Busetta, N. Howden, R. Ronnquist, and A. Hodgson. Structuring BDI Agents in Functional Clusters. In N. R. Jennings and Y. Lespérance, editors, *Intelligent Agents VI. Agent Theories, Architectures, and Languages - 6th International Workshop*, number 1757 in Lecture Notes in Artificial Intelligence, Orlando, FL, USA, July 1999. Springer-Verlag.

[13] B. Chandrasekaran, J. R. Josephson, and V. R. Benjamins. What Are Ontologies, and Why Do We Need Them. *IEEE Intelligent Systems*, 14(1):20–26, January/February 1999.

[14] S. H. Clearwater, R. Costanza, M. Dixon, and B. Schroeder. Saving Energy Using Market-Based Control. In *Marked Based Control*, pages 253–273. World Scientific: Singapore, 1996.

[15] M. Coburn. *JACK Intelligent Agents User Guide.* Agent Oriented Software Ltd., Melbourne, Australia, October 2000.

[16] P. L. Guida (Project Coordinator). EuROPE–TRIS Project — 4th European Union RDT Program Telematics Applications Programme, 1998. Transport (Rail) Contract DGXIII - TR 1022 (TR), Specifications and System Design, Deliverable D04.2. Web page: http://www.srd.it/TRISprj.

[17] J. M. Corera, I. Laresgoiti, and N. R. Jennings. Using ARCHON, Part 2: Electricity Transportation Management. *IEEE Expert*, 11(6):71–79, 1996.

[18] S. Crocè. Un approccio ad agenti per la simulazione di un modello postale. Master's thesis, DISI – Università di Genova, Genova, Italy, 1998. In Italian.

[19] A. Cuppari, P. L. Guida, M. Martelli, V. Mascardi, and F. Zini. An Agent Based Prototype for Freight Trains Traffic Management. In *Proc. of FMERail'99 Workshop (a satellite event of FM'99)*, Toulouse, France, September 1999.

[20] A. Cuppari, P.L. Guida, M. Martelli, V. Mascardi, and F. Zini. Prototyping Freight Trains Traffic Management Using Multi-Agent Systems. In *Proc. of IEEE International Conference on Information, Intelligence and Systems*, Washington, DC, November 1999. IEEE Computer Society Press.

[21] P. Dart, E. Kazmierckaz, M. Martelli, V. Mascardi, L. Sterling, V.S. Subrahmanian, and F. Zini. Combining Logical Agents with Rapid Prototyping for Engineering Distributed Applications. In *Proc. 9th International Conference of Software Technology and Engineering (STEP'99)*, Pittsburgh, PA, USA, September 1999. IEEE Computer Society Press.

[22] R. Degl'Innocenti. UML come Linguaggio per Specificare Agenti: Analisi, Estensione ed Integrazione in CaseLP. Master's thesis, DISI - Università di Genova, Genova, Italy, 2000. In Italian.

[23] G. Delzanno. *Logic & Object-Oriented Programming in Linear Logic.* PhD thesis, Università di Pisa, Dipartimento di Informatica, March 1997. Technical Report TD 2/97.

[24] G. Delzanno and M. Martelli. Proofs as Computations in Linear Logic. *Theoretical Computer Science*. To appear.

[25] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A Formal Specification of dMARS. In M. P. Singh, A. Rao, and M. Wooldridge, editors, *Intelligent Agents IV. Agent Theories, Architectures, and Languages - 4th International Workshop*, number 1365 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1997.

[26] R. Doorenbos, O. Etzioni, and D. Weld. A Scalable Comparison-Shopping Agent for the World Wide Web. In *Proc. of the 1st International Conference on Autonomous Agents (Agents'97)*, pages 39–48, Marina del Rey, CA, USA, 1997.

[27] C. Elliott and J. Brzezinski. Autonomous Agents as Synthetic Characters. *AI Magazine*, pages 13–29, Summer 1998.

[28] H. E. Eriksson and M. Penker. *ULM Toolkit.* John Wiley & Sons, 1998.

[29] M. Fisher. Representing Abstract Agent Architectures. In M. P. Singh J. P. Müller and A. S. Rao, editors, *Intelligent Agents V. Agent Theories, Architectures, and Languages - 5th International Workshop*, number 1555 in Lecture Notes in Artificial Intelligence, pages 227–241, Paris, France, July 1998. Springer-Verlag.

[30] S. Franklin and A. Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In J. P. Müller, M. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III. Agent Theories, Architectures, and Languages - 3rd International Workshop*, number 1193 in Lecture Notes in Artificial Intelligence, Budapest, Hungary, August 1996. Springer-Verlag.

[31] M. R. Genesereth and S. P. Ketchpel. Software Agents. *Communications of the ACM*, 37(7):49–53, July 1994.

[32] G. De Giacomo, Y. Lespérance, and H. J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[33] J. Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[34] S. Grand and D. Cliff. Creatures: Entertainment Software Agents with Artificial Life. *Autonomous Agents and Multi-Agent Systems*, 1(1), 1998.

[35] Programming Systems Group. *SICStus Prolog 3.8.5 User's Manual*. Swedish Institute of Computer Science, Kista, Sweden, September 2000.

[36] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J. C. Meyer. Control Structures of Rule-Based Agent Languages. In M. P. Singh J. P. Müller and A. S. Rao, editors, *Intelligent Agents V. Agent Theories, Architectures, and Languages - 5th International Workshop*, number 1555 in Lecture Notes in Artificial Intelligence, pages 381–396, Paris, France, July 1998. Springer-Verlag.

[37] C. A. Iglesias, M. Garijo, and J. C. González. A Survey of Agent-Oriented Methodologies. In M. P. Singh J. P. Müller and A. S. Rao, editors, *Intelligent Agents V. Agent Theories, Architectures, and Languages - 5th International Workshop*, number 1555 in Lecture Notes in Artificial Intelligence, Paris, France, July 1998. Springer-Verlag.

[38] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An Architecture for Real-Time Reasoning and System Control. *IEEE Expert*, 7(6), 1992.

[39] N. R. Jennings, K. Sycara, and M. Wooldridge. A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems*, 1:7–38, 1998.

[40] D. Kinny, M. Georgeff, and A. Rao. A Methodology and Modelling Technique for System of BDI Agents. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away*, number 1038 in Lecture Notes in Artificial Intelligence, pages 56–71. Springer-Verlag, 1996.

[41] R. A. Kowalski and F. Sadri. From Logic Programming Towards Multi-agent Systems. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):391–419, 1999.

[42] J. Kramer. Distributed Software Engineering. In B. Fadini, editor, *Proc. of the 16th International Conference on Software Engineering*, pages 253–266, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[43] Y. Labrou and T. Finin. A Proposal for a new KQML Specification. Technical Report TR CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD, 1997.

[44] Y. Labrou, T. Finin, and Y. Peng. Agent Communication Languages: the Current Landscape. *IEEE Intelligent Systems*, pages 45–52, March/April 1999.

[45] M. Ljunberg and A. Lucas. The OASIS Air Traffic Management System. In *Proc. of the 2nd Pacific Rim International Conference on AI (PRICAI'92)*, Seoul, Korea, 1992.

[46] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[47] H. Lu, L. Sterling, and A. Wyatt. SportsFinder: An Information Agent to Extract Sports Results from the World Wide Web. In D. Ndumu and H. Nwana, editors, *Proc. of The Fourth International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'99)*, London, UK, April 1999.

[48] J. Lu, A. Nerode, and V. S. Subrahmanian. Hybrid Knowledge Bases. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):773–785, October 1996.

[49] P. Maes. Agents that Reduce Work and Information Overload. *Communications of the ACM*, 37(7):31–40, July 1994.

[50] S. Marcus and V. S. Subrahmanian. Foundations of Multimedia Database Systems. *Journal of the ACM*, 43(3):474–523, 1996.

[51] S. Marini. Specifica di Sistemi Multi-Agente Eterogenei. Master's thesis, DISI - Università di Genova, Genova, Italy, 1999. In Italian.

[52] S. Marini, M. Martelli, V. Mascardi, and F. Zini. HEMASL: A Flexible Language to Specify Heterogeneous Agents. In A. Corradi, A. Omicini, and A. Poggi, editors, *WOA 2000. Dagli Oggetti agli Agenti*, pages 76–81, Parma, Italy, May 2000.

[53] S. Marini, M. Martelli, V. Mascardi, and F. Zini. Specification of Heterogeneous Agent Architectures. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures, and Languages - 7th International Workshop*, Lecture Notes in Artificial Intelligence, Boston, MA, USA, July 2000. Springer-Verlag.

[54] M. Martelli, V. Mascardi, and F. Zini. Applying logic programming to the specification of complex applications. *Mathematical Modelling and Scientific Computing*, 8, 1997.

[55] M. Martelli, V. Mascardi, and F. Zini. CaseLP: a Complex Application Specification Environment Based on Logic Programming. In *Proc. of ICLP'97 Post Conference Workshop on Logic Programming and Multi-Agents*, pages 35–50, Leuven, Belgium, July 1997.

[56] M. Martelli, V. Mascardi, and F. Zini. Towards Multi-Agent Software Prototyping. In H. S. Nwana and D. T. Ndumu, editors, *Proc. of The 3rd International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'98)*, pages 331–354, London, UK, March 1998.

[57] M. Martelli, V. Mascardi, and F. Zini. Specification and Simulation of Multi-Agent Systems in CaseLP. In M. C. Meo and M. Vilares Ferro, editors, *Proc. of Appia–Gulp–Prode 1999*, L'Aquila, Italy, September 1999.

[58] V. Mascardi. An Agent-Based Approach to Distributed Simulation, Distributed Prototyping and Software Integration, 1999. PhD thesis proposal, DISI – Università di Genova, Genova, Italy.

[59] V. Mascardi and E. Merelli. Agent-Oriented and Constraint Technologies for Distributed Transaction Management. In *Proc. of IIA and SOCO*, Genova, Italy, June 1999. ICSC Academic Press.

[60] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an Agent Communication Language. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II. Agent Theories, Architectures, and Languages - 2nd International Workshop*, number 1037 in Lecture Notes in Artificial Intelligence, Montreal, Canada, August 1995. Springer-Verlag.

[61] D. Miller. Forum: A Multiple-Conclusion Specification Logic. *Theoretical Computer Science*, 165(1), 1996.

[62] M. Mulder, J. Treur, and M. Fisher. Agent Modelling in METATEM and DESIRE. In M. P. Singh, A. Rao, and M. Wooldridge, editors, *Intelligent Agents IV. Agent Theories, Architectures, and Languages - 4th International Workshop*, number 1365 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1997.

[63] J. P. Müller. Control Architectures for Autonomous and Interacting Agents: a Survey. In L. Cavedon, A. S. Rao, and W. Wobcke, editors, *Intelligent Agent Systems: Teoretical and Pratical Issues*, number 1209 in Lecture Notes in Artificial Intelligence, Cairns, Australia, 1997. Springer-Verlag.

[64] J. P. Müller. The Right Agent (Architecture) to Do the Right Thing. In M. P. Singh J. P. Müller and A. S. Rao, editors, *Intelligent Agents V. Agent Theories, Architectures, and Languages - 5th International Workshop*, number 1555 in Lecture Notes in Artificial Intelligence, pages 211–225, Paris, France, July 1998. Springer-Verlag.

[65] P. Naur and B. Randell, editors. *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee*. NATO, 1969.

[66] D. T. Ndumu and H. S. Nwana. Research and Development Challanges for Agent-Based Systems. *IEE Proc. of Software Engineering*, 144(1):2–10, February 1997.

[67] I. Noda, S. Suzuki, H. Matsubara, M. Asada, and H. Kitano. RoboCup-97: The First Robot World Cup Soccer Games and Conference. *AI Magazine*, pages 49–59, Fall 1998.

[68] M. De Pedrini. CaseLP Visualizer: un Tool di Visualizzazione per Sistemi Multi-Agente Logici. Master's thesis, DISI – Università di Genova, Genova, Italy, 1998. In Italian.

[69] A. Perini, F. Giunchiglia, J. Mylopoulos, P. Bresciani, and P. Giorgini. A Knowledge Level Software Engineering Methodology for Agent Oriented Software. Technical Report IRST 0011-01, Istituto Trentino di Cultura, Povo, Trento, Italy, November 2000.

[70] G. Persano. Gestione Distribuita di Informazioni Mediche Mediante Tecniche Multi-Agente. Master's thesis, DISI – Università di Genova, Genova, Italy, 1998. In Italian.

[71] R. S. Pressman. *Software Engineering. A Practitioner's Approach.* McGraw–Hill International, UK, 3rd edition, 1994. European Edition. Adapted by D. Ince.

[72] A. Rao and R. Georgeff. Modeling Rational Agents within a BDI–Architecture. In R. Fikes and E. Sandewall, editors, *Proc. of Knowledge Representation and Reasoning (KR&R-91)*, pages 473–484, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[73] A. S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away*, number 1038 in Lecture Notes in Artificial Intelligence, pages 42–55. Springer-Verlag, 1996.

[74] M. Schroeder and G. Wagner. Vivid Agents: Theory, Architecture, and Applications. Submitted to the Journal of Logic and Computation.

[75] Y. Shoham. Agent-Oriented Programming. *Artificial Intelligence*, 60(1):51–92, March 1993.

[76] Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1-3):17–64, October/December 1996.

[77] M. Spivey. *The Z Notation (second edition).* Practice Hall International, 1992.

[78] L. Sterling. On finding needles in WWW haystacks. In *Proc. of the 10th Australian Joint Conference on Artificial Intelligence*, number 1342 in Lecture Notes in Artificial Intelligence, Perth, Australia, 1997. Springer-Verlag.

[79] L. Sterling, P. Ciancarini, and T. Turnidge. On the Animation of "not Executable" Specifications by Prolog. *International Journal of Software Engineering and Knowledge Engineering*, 6(1):63–87, 1996.

[80] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1994.

[81] V. S. Subrahmanian. *Principles of Multimedia Database Systems.* Morgan Kaufman Press, January 1998.

[82] M. Tsvetovatyy, M. Gini, B. Mobasher, and Z. Wieckowski. MAGMA: an Agent-Based Virtual Marketplace for Electronic Commerce. *Applied Artificial Intelligence*, 11(6):501–524, 1997.

[83] M. H. van Hemden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23:733–742, 1976.

[84] H. Wang and C. Wang. Intelligent Agents in the Nuclear Industry. *IEEE Computer*, 30(11):28–34, 1997.

[85] R. Weihmayer and H. Velthuijsen. Intelligent Agents in Telecommunications. In *Agent Technology: Foundations Applications and Markets*, pages 203–217. Springer-Verlag, Berlin, Germany, 1998.

[86] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[87] E. Yu. *Modeling Strategic Relationships for Process Reengineering.* PhD thesis, University of Toronto, Department of Computer Science, Toronto, Canada, October 1995.

[88] F. Zini and L. Sterling. Designing Ontologies for Agents. In M. C. Meo and M. Vilares Ferro, editors, *Proc. of Appia–Gulp–Prode 1999*, L'Aquila, Italy, September 1999.