CaseLP: a Prototyping Environment for Heterogeneous Multi-Agent Systems

V. Mascardi (mascardi@disi.unige.it)
DISI - Università di Genova
Via Dodecaneso 35, 16146, Genova, Italy.

M. Martelli (martelli@disi.unige.it)
DISI - Università di Genova
Via Dodecaneso 35, 16146, Genova, Italy.

F. Zini (zini@itc.it)
ITC-irst
Via Sommarive 18, 38050 Povo (Trento), Italy.

R. Degl'Innocenti (roberto.deglinnocenti@elsag.it)
Elsag S.p.A.
Via Puccini 2, 16154, Genova, Italy.

Abstract. This paper describes CaseLP, an environment for the rapid prototyping of Multi-Agent Systems (MASs). CaseLP provides the developer with heterogeneous languages for the specification, design and implementation of the MAS components and their interactions, and an engineering methodology for the realization of the prototype following a sequence of clear steps. CaseLP agents are characterized by their architecture, the roles they play in the MAS, the ontologies they understand, their internal program, and their state. CaseLP offers high-level languages for modeling all these aspects, as well as semi-automatic compilers for implementing executable code starting from the heterogeneous specifications of the agents' components and tools for running and debugging the resulting prototype. The high-level specification languages range from rule-based ($\mathcal{E}_{hhf}$, AgentRules) to graphical (an agent-oriented extension of UML) to object-oriented (HEMASL). One specification language, $\mathcal{E}_{hhf}$, is executable, allowing the early verification and testing of the specification. The prototype implementation language is based on Prolog, and a graphical user interface is provided to help the MAS developer to iteratively test, debug, and refine design and implementation choices. CaseLP has been proven useful for engineering MAS prototypes in application domains ranging from vehicle monitoring to remote video-encoding of mail pieces, from distributed health-care management to decision support systems. Many of these applications were developed for industrial partners. The paper also discusses in details some methodologies and frameworks for MAS development and compares them with CaseLP.

Keywords: Agent-oriented software engineering, MAS rapid prototyping, heterogeneous agents.

1. Introduction

It's four a.m. of a working day in 2010: trading in the fruit market begins. A couple of brokers helps buyers in finding which sellers sell the fruit they may be interested in and traders frantically negotiate the best price to sell or buy goods. The amount exchanged goods among traders is impressive, driven by sophisticated negotiation strategies. Tons of fruit change owner every few milliseconds...

We start our paper with this snapshot of close-future life. The fruit market we are referring to is an electronic market-place and the traders and brokers are software entities, which act on behalf of their human owners by autonomously carrying out the negotiation and the exchange of fruit. The traders and brokers are distributed, each one connecting to the virtual market-place from a different location, and are heterogeneous, each of them having been developed by different people, using different software applications and languages. They are able to make decisions, to communicate with other entities in the market-place, to sense changes occurring in their environment and to react to them in real time.

The correct and efficient engineering of real-world applications characterized by the features above is the main goal of *Agent Oriented Software Engineering (AOSE)*, a very active research area that has been working for finding abstractions, languages, methodologies and toolkits to be used for their modeling verification, validation and implementation. The metaphor underlying AOSE is that the components of complex real-world applications are *intelligent agents*, which interact and exchange information in a society (or *Multi-Agent System*), collaborating for reaching a common target, or competing to maximize their personal profit. An *intelligent agent*, according to a classical definition proposed by N. R. Jennings, K. Sycara and M. Wooldridge in [47], is

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

Despite AOSE techniques have been successfully used for the development of a notable number of applications in several fields, researchers and practitioners agree that MASs are very complex systems and the engineering of an end product as a MAS involves a non negligible degree of risk. For this reason we think that the most suitable approach to engineer a MAS is to build a working prototype, in order to validate, test and possibly modify the initial requirements of the MAS to be developed before its actual realization.

Compared to other software engineering paradigms, prototyping gives three main advantages in software realization: *rapidity*, *low cost*, and *iterative development*. The amount of time required to develop a working prototype is usually limited. This means the early availability of a product to be evaluated by the customer, allowing 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 economic risks concerning the development process. Finally, prototyping is an iterative process and this gives the flexibility of revising requirements or critical design choices several times before committing to some final decisions. Another advantage is that a prototype usually has not to be very efficient and therefore it can be realized using formal methods and tools. This generally allows to realize a prototype of a software product following a quite formal approach, even if it is not feasible to adopt the same formal techniques to realize the final version of the product. If the prototyping framework allows for the integration of external packages that will be part of the final application inside the prototype itself, the prototype becomes extremely close to the end product and may provide realistic information on its behavior. Moreover, if the prototyping environment allows agents specified in different languages to be seamlessly integrated within the same prototype, reuse of already specified components is facilitated, making the prototyping process even faster.

The question which could be raised at this point is: “Are there any tools and methodologies which may help the software engineer to develop heterogeneous, multi-language prototypes where it is possible to integrate external software?”

In this paper we provide a possible answer to this question by introducing ARPEGGIO, an open approach for the specification, rapid prototyping and engineering of heterogeneous agent-based software, and by discussing in details CaseLP, an instance of ARPEGGIO. CaseLP has been developed in the last few years and used to model and implement prototypes of MASs in different domains. This paper is the first work where we systematically describe our experience with CaseLP.

CaseLP (*Complex Applications Specification Environment based on Logic Programming*) was initially born as a language for extending a constraint logic programming language with theories, communication predicates and “safe” state update predicates (namely, without permanent effects in case of failure) [57],[58]. Using the preliminary version of CaseLP (which, at that time, was named ACLPL, *Agent Constraint Logic Programming Language*) we developed the application described in Section 8.1 in collaboration with the Elsag S.p.A. international company. This experience was so encouraging that we decided to invest more efforts on CaseLP, consolidating its implementation, testing our improvements by developing an application with the Italian Railway Company [96], and giving more attention to the integration of external software [59]. We soon realized that, in order to best exploit the CaseLP tool, we needed to develop a methodology for MAS prototypes development.

During the year 1998 we started working in this direction [60], and in the same year we developed a graphical user interface for our environment [78]. Our following step was to integrate an executable linear-logic based language, $\mathcal{E}_{hhf}$, into our environment This was first done in [3], which demonstrated the suitability of linear logic for modeling agents with heterogeneous architectures, and for easily modeling the evolution of a MAS. The work done in [3] was refined in [9, 10], were we also showed that it was possible to semi-automatically translate $\mathcal{E}_{hhf}$ specifications of agents into extended Prolog code. In the meanwhile, we went on developing applications in different domains [65, 80, 21, 22, 23], and we refined both the methodology and the prototyping tools [61, 62].

During the year 1999, together with the Logic Programming Group at the Computer Science Department of the University of Maryland, USA, and the Logic Programming and Software Engineering Group at the Computer Science & Software Engineering Department of the University of Melbourne, Australia, we started to define the ARPEGGIO approach [24] discussed in the next section. In the same year, we started extending CaseLP by adding non-logical languages to the set of supported prototyping languages. For this purpose we designed a procedural language, HEMASL [54], for the specification of both the agent architecture and its program and state.

Following the current research trends in AOSE, in the year 2000 we started exploring howUML could be extended to model agent features and how the extended UML could be integrated in CaseLP [25]. During this year we refined the HEMASL language [55, 56] and we developed other applications of industrial interest with Elsag S.p.A. [2] and with the Italian Railway Company [94].

This paper discusses the state of the art of the CaseLP project; in the meantime CaseLP has been further developed up to the design and initial implementation of a distributed prototyping environment, D-CaseLP[1, 67, 64], outlined in the last section. The paper is structured as follows: Section 2 discusses the ARPEGGIO open approach, the rationale behind it, and introduces CaseLP as an instance of ARPEGGIO; Section 3 discusses the CaseLP agent model; Section 4 describes the hierarchy of abstraction levels we have defined to easy MAS modeling; Section 5 discusses the CaseLP development methodology while Section 6 gives an overview of tools and languages provided by the CaseLP environment; Section 7 shows the use of CaseLP methodology, tools, and languages through an example: the specification and and implementation of a distributed fruit market; Section 8 describes some applications developed with CaseLP; Section 9 discusses related work and, finally, Section 10 concludes the paper and suggests further directions of research.

2. The ARPEGGIO open approach

ARPEGGIO (*Agent based Rapid Prototyping Environment Good for Global Information Organization* [24]) is an approach that aims at the realization of an open framework based on logic programming for the rapid prototyping of heterogeneous multi-agent systems.

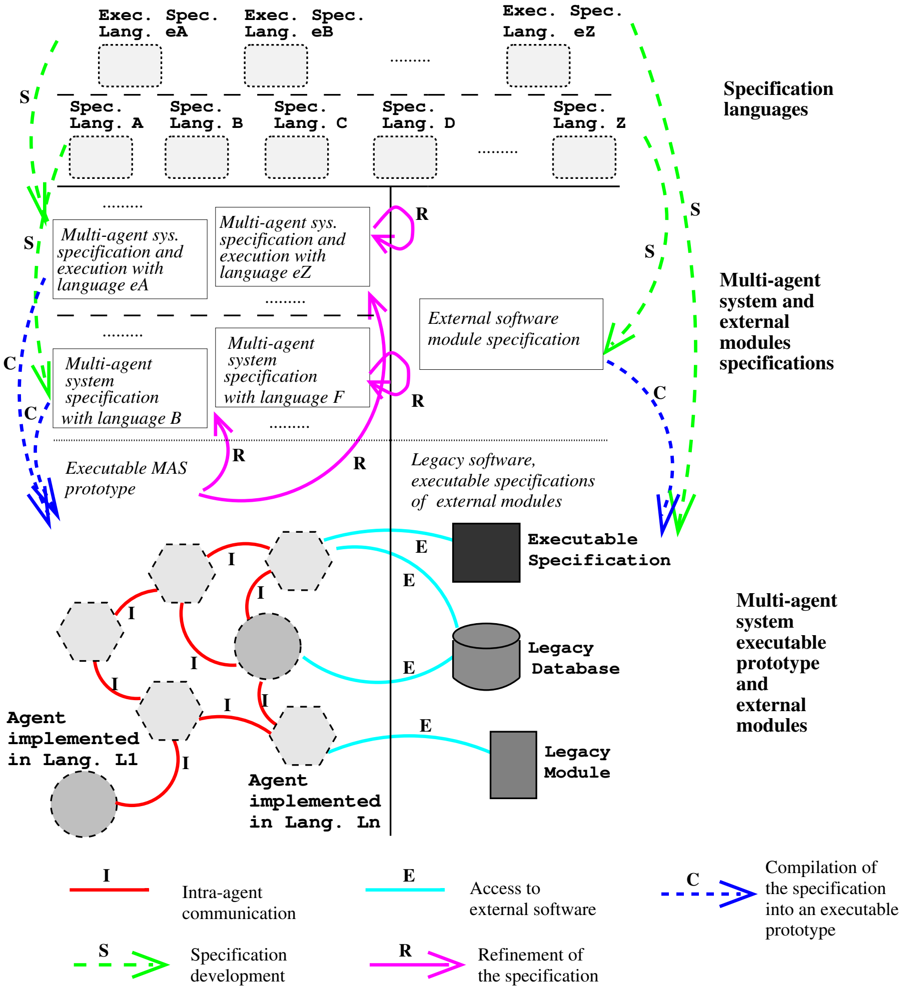


Figure 1. The ARPEGGIO open approach.

The philosophy behind ARPEGGIO is sketched in Figure 1. We assume to have at our disposal a collection of heterogeneous specification languages, depicted at the top of the picture. Each language is suitable for model a particular aspect of the MAS prototype architecture and/or behavior.

The development of a specification is described in the picture by the arrows labeled *S*. The top layer in Figure 1 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 labeled *C* on the left of the picture represent this compilation step.

If an executable specification language is 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 as expected. In Figure 1, the arrows labeled *R* represent the refinement stage. Once the software engineer is confident enough in the specification of the system, it may be compiled, obtaining a system closer to the final application that incorporates the legacy components.

The bottom rightmost part of Figure 1 represents legacy software and executable specifications of external modules that are integrated into the prototype. The access to external software is represented by the lines labeled*E*. Even if the purpose of ARPEGGIO is to support the development of MAS *prototypes*, and not end products, we think that the possibility to integrate external legacy software into the prototype, to run the "extended" prototype, and to test its behavior may allow the software engineer to develop the final product in less time, with less cost, and with a greater confidence on the expected final behavior.

The main advantage in this approach s that developers that want to prototype applications in ARPEGGIO do not necessarily need to be "experts" in all the specification languages but can choose the most useful ones with respect to their needs. For example, a non-executable specification language `A` could be used to model the high level interaction protocols between agents, while an executable specification language `eC` could be suitable for specifying the global evolution of the system. A specification language `D` could be suitable for describing the static structure of the agents, while another executable specification language `eB` could help in modeling the dynamic behavior of a single agent.

ARPEGGIO takes contributions from three international Logic Programming research groups: the LP Group at the Computer Science Department of the University of Maryland, USA; the LP and SE Group at the Computer Science & Software Engineering Department of the University of Melbourne, Australia; and the LP Group at the Computer Science Department of the University of Genova, Italy.

CaseLP, developed by the last research group, is intended as a contribution to the diffusion of ARPEGGIO . It implements ARPEGGIO concepts mainly in the methodological aspects of modeling and implementing the prototype as well as in the integration of different languages for the specification and implementation of an executable prototype. In fact, CaseLP contributes to ARPEGGIO in the following way:

– it provides the prototype developer with a set of heterogeneous high-level specification languages: HEMASL, UML, $\mathcal{E}_{hhf}$, CaseLP-Protocol, CaseLP-Ontology, AgentOntology, AgentRules, CaseLP-MASadl; each of these languages is suitable for specifying one or more aspects of the MAS structure and dynamics;

- one of the high-level specification languages, $\mathcal{E}_{hhf}$, can be executed to early verify properties of the MAS;
- there is one target language, ProlAg, into which the high-level specifications can be compiled in order to be executed either under the CLP language ECLiPSE [33] or under SICStus Prolog [89];
- semi-automatic mechanisms for translating agent specifications from $\mathcal{E}_{hhf}$ and AgentRules into ProlAg are given;
- there are external languages and packages which can be integrated in the working prototype, namely those for which ECLiPSE and SICStus Prolog provide an interface.

3. CaseLP agent model

3.1. A DESCRIPTIVE PERSPECTIVE...

From a descriptive point of view, a CaseLP agent is characterized by three orthogonal features:

- its **architecture**;
- the **roles** it can play;
- the **ontologies** it knows.

The agent's *architecture* is determined by both the agent internal data structures and the control flow among them, which we call the *engine* of the architecture. Paraphrasing N. Wirth's famous equation,

Algorithms + Data Structures = Programs

we could say that

$\boxed{\text{Engine + Internal Data Structures = Architecture}}$

The architecture is a *domain independent* feature: agents with the same internal data structures and behaving according to the same engine can be adopted in very different domains, provided that they have different "domain dependent knowledge". This domain knowledge comes from the data structures *content*, which can dramatically vary from agent to agent, and by the *ontologies* the agent knows. M. J. Wooldridge and N. R. Jennings' survey

[92] provides a comprehensive review of architectures and their applications which confirms their domain independent nature.

As far as roles are concerned, the notion of an agent's *role*

> focuses on the position and responsibilities of an object within an overall structure or system [51].

For M. J. Wooldridge, N. R. Jennings and D. Kinny [93]

> ... a role is defined by four attributes: responsibilities, permissions, activities, and protocols.

In our CaseLP descriptive model, a role definition is definitely simpler than the ones we quoted, since it is characterized by the set of services required and provided by an agent:

$\boxed{\text{Required Services + Provided Services = Role}}$

Each service may be required (provided) following communication protocols with agents providing (requiring) complementary services. A single service may need the exchange of many messages to be supplied, thus the services an agent provides and the messages it is able to send are not in one-to-one correspondence. Agent roles, like agent architectures, are *domain independent* features: let us think of a *seller* agent which provides the *sell* service. This is a domain independent service, since it abstracts from the sold merchandise and just deals with the ability of the agent to sell something.

Finally, *ontologies* are structures that hold information about entities, their properties, and relationships among them that are possible in a specific domain of knowledge [17]. In knowledge-based systems, an ontology is that part of the system which specifies what things exist and what is true about them. Ontologies are used and represented in CaseLP in a standard way. We may summarize our model by saying that

$\boxed{\text{Concepts + Properties of concepts + Relations among them = Ontology}}$

Ontologies may be either *domain independent* or *domain dependent* features: some ontologies establish concepts and relations necessary for inter-domain integration, for example those characterizing agents, roles, protocols, architectures and the constraints which must be respected when a MAS is built. Other ontologies are more specific, and give the meaning to concepts used only in a particular domain. For example, an ontology named fruit specifying that "apple" is a non-exotic fruit with certain properties instead of a computer trade-mark, is useful for a seller agent working in an electronic fruit marketplace, but it is misleading for a seller agent trading in computers.

Roles, architectures and ontologies are orthogonal concepts: the same role can be played by agents with different architectures, and agents characterized by the same architecture can play different roles. Agents with the same architecture and roles, but knowing different ontologies, will behave in different ways: a fruit seller and a computer seller will give different meanings to the same terms, and will then adopt different strategies for satisfying the requests they receive, according to the ontologies they refer to. Roles are related with the specification of the MAS at the macro-level, namely at the level where only the interactions among agents are of interest, abstracting from their internal functioning. The aspect of internal functioning is dealt with when specifying agents architectures, which provide the internal framework for allowing agents to reason, to decide which actions should be taken and to interact with other agents and with the environment. These decisions are influenced by the ontologies which operate at the knowledge-level for organizing and structuring the agents' knowledge.

3.2. . . . AND A COMPUTATIONAL PERSPECTIVE

From a computational point of view, the agents we model in CaseLP are characterized by

– a **state**,

– a **program** and

– an **engine**.

The *state* includes data that may change during the execution of the agent. For example, the state of a BDI agent [83] contains its beliefs, goals, intentions and an event queue.

The *program* contains information that does not change during the execution of the agent. For example, the program of a BDI agent is determined by its plans; they describe the actions the agent should take according to its current beliefs and the content of the event queue. The actions an agent is able to perform vary from architecture to architecture, but usually an agent is at least capable of updating its state and interacting with the environment and with other agents in the MAS by sending and receiving messages.

Lastly, the *engine* controls the execution of the agent. To go on with the BDI example, the typical BDI engine is characterized by the following actions:

1. observe the world and the agent's internal state, and update the *event queue* consequently;

2. generate possible new plans whose trigger event matches an event in the event queue and whose context is satisfied;

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

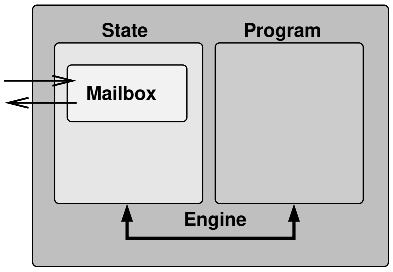


Figure 2. General CaseLP agent architecture.

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.

The engine and the program belong to different abstraction levels: the engine is a meta-interpreter for the program and the data (state) on which the program operates. The behavior of the agent is determined by the application of the agent program on the agent state by means of the agent engine.

The architecture of an agent is characterized by components which contain its state, components for the program, and an engine operating on them. The content of the agent components will be expressed using some architecture dependent object language for which the engine provides an interpreter. We do not commit to any object language, since our model aims at being as general as possible. The only assumption we make is on the state components: we assume that there is a *mailbox*[1] among them which allows asynchronous communication among agents and between the agents and the external environment. Every CaseLP agent has an architecture which specializes the one depicted in Figure 2.

Agents which share this general architecture can be described in CaseLP at different levels of abstraction, following a methodology which guides the developer from the informal description of the MAS to the development of a working prototype. In each step of the method the user can adopt the most suitable tools and languages among the provided ones. The next sections deal with these three topics respectively: hierarchy of abstraction levels (Section 4), development methodology (Section 5) and tools and languages provided by CaseLP (Section 6).

[1] We name this component *mailbox* but it should be named *event queue*, like in the BDI architecture, or with other equivalent denominations.

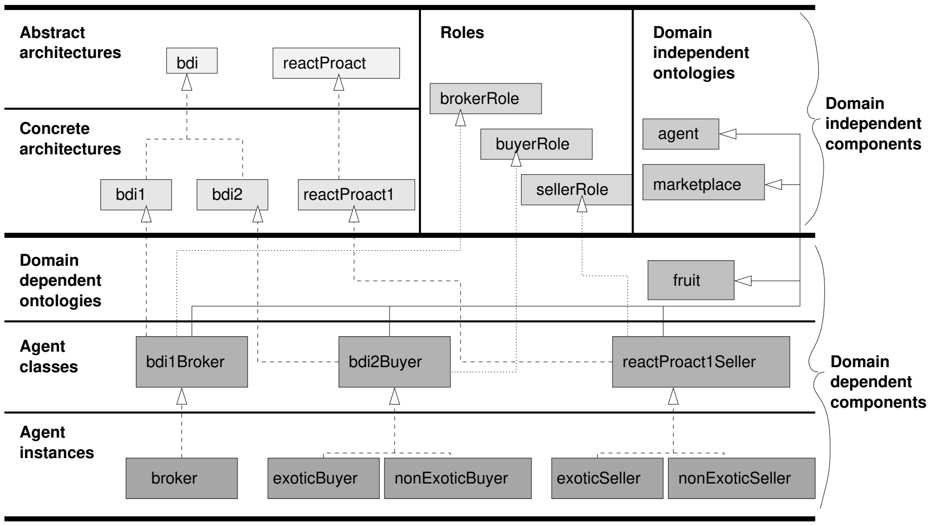


Figure 3. Abstraction hierarchy.

4. Hierarchy of abstraction levels

In order to ensure modularity, flexibility and reusability in the conceptualization and definition of heterogeneous agents, we have introduced an abstraction hierarchy illustrated in Figure 3 by means of the fruit marketplace example. It is possible to note two main horizontal layers of components in this hierarchy: the domain independent components and the domain dependent ones. We have already introduced the components we consider to be domain independent, namely

- architectures;
- roles and
- domain independent ontologies.

The components which include domain dependent knowledge are

- classes of agents;
- instances of agents and
- domain dependent ontologies.

The rationale behind keeping the domain dependent and independent components clearly separate lies upon *reusability* and *separation of competencies among developers*. The domain independent components, once specified or

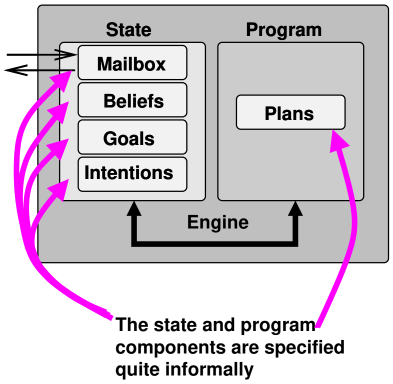


Figure 4. Abstract BDI architecture.

implemented, can be reused for building prototypes of many different applications: roles, architectures and domain independent ontologies can be defined once and reused forever. Besides this, it is conceptually correct to think that the development of a MAS prototype will involve different people with different skills. There will be the expert of architectures, the expert of roles, the expert of ontologies, and there will be the domain experts, which know how to implement the classes and instances of agents. The clear separation of the various features to be specified to develop a MAS facilitates this separation of competencies.

In the following sections we will concentrate on abstract and concrete architectures and on classes and instances of agents. Their definition can be obtained by progressively filling and refining the content of the components belonging to the general architecture depicted in Figure 2. Our characterization of ontologies and roles is quite simple and does not need to be furtherly expanded. We just explain the meaning of the ontologies depicted in Figure 3: the domain independent agent ontology [95] structures the knowledge about agents by specifying, for example, that each agent class must play at least one role and have exactly one concrete architecture. The marketplace ontology, described in [25] and shown in Section 7.1.5, establishes the meaning of a "contract" and of the "merchandise" to be exchanged. The fruit ontology establishes which entities are fruits, and which fruits are exotic or not.

4.1. Abstract architecture

The abstract architecture defines the components of the architecture and the basic structure of the engine. It is possible to provide guidelines for the realization of any step of the engine without necessarily giving all the implementation details. In Figure 3 there are two abstract architectures, the BDI one (named bdi in the figure) and the reactive–proactive one (named react-

Proact)[2]. The abstract architecture for a BDI agent, sketched in Figure 4, could be characterized by

- a component for beliefs,
- a component for goals,
- a component for intentions and
- a component for plans.

We do not need to explicitly say that an event queue is needed, because our general agent architecture already contains this kind of structure, which we have named the mailbox. Moreover, at this stage we do not need to assign a type to the components characterizing the architecture, since these details will be provided when implementing the concrete architecture.

4.2. CONCRETE ARCHITECTURE

A concrete architecture is defined starting from an abstract architecture:

- each component is assigned a type among the ones the chosen specification language provides;
- each step of the engine is fully specified.

For example, it should be the case that each intention in one implementation of the bdi abstract architecture, say bdi1, be a stack of strings, and the intention component be a list of such intentions. Another concrete implementation of the bdi architecture, bdi2, could implement the intention component as a set of lists of tuples: despite the differences between this architecture implementation and the previous one, both of them are BDI architectures and it is reasonable that both of them specialize the same abstract architecture. Besides differing in the type assigned to the architecture components, bdi1 and bdi2 could also differ in the engine implementation, which can specialize the specification given in the previous abstraction level in different ways.

4.3. AGENT CLASS

A class (Figure 5) is defined by instantiating the components which contain the program of the agent in the concrete architecture, according to the role the class plays. Any agent class playing a role must adhere to the specification of that role, namely, it must require and provide the services specified by the

[2] A reactive–proactive architecture is characterized by a rule-based program whose rules can be fired either by the reception of an external message (reaction) or by the verification of a condition over the agent's state (proaction).

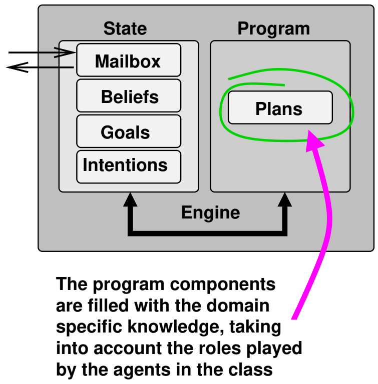


Figure 5. A class of BDI agents.

role. In other words, if we consider the specification of a role as an interface definition, a class playing that role must implement the role interface. A class can play one or more roles; this means that the instances of that class have the *potentiality* to play these roles during their life. It depends on the internal evolution, which in turn depends on the overall MAS evolution, whether an agent will switch form a role to another one during its life or not.

Continuing the fruit market example, the application to be developed could need a broker agent adhering to the bdi1 concrete architecture, which operates in the context of a market. The corresponding class, bdi1Broker, will be characterized by the bdi1 plans describing the behavior of a market broker agent as specified by the brokerRole. Other necessary classes could be a bdi2 buyer class, bdi2Buyer, and a reactProact1 seller, reactProact1Seller[3].

4.4. Agent instance

Starting from an existing class, the initialization of the architecture components containing the state identifies an agent instance (Figure 6). To complete our simple running example let us consider the instances of fruit buyers implementing the bdi2 concrete architecture. If a buyer is characterized by its beliefs about its available budget, the kind of fruit it is willing to buy, the price it is willing to pay for the fruit and the current amount of stored fruit, this information can be stored in the Beliefs component of the agent state, letting the possibility to define the bdi2Buyer class only once, and deriving whatever number of buyer instances from it by specifying, for each instance, the values of these three features. In our marketplace example there are an

[3] The choice of the BDI architecture for agents playing broker and buyer roles and of the reactive–proactive architecture for sellers has no rationale behind apart from illustrating CaseLP abstraction hierarchy. In real applications guidelines like the ones discussed in [69] should be followed to identify "the right agent architecture to do the right thing".

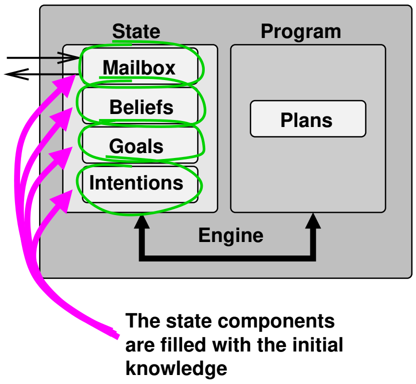


Figure 6. An instance of a BDI agent.

exotic fruit buyer, exoticBuyer, willing to buy pineapples, passion fruits, bananas and so on, and a non-exotic fruit buyer, nonExoticBuyer, who wants to buy apples, pears, peaches, plums, strawberries. The Goals, Intentions and Mailbox components of both instances are initially empty.

This example should clarify the usefulness of ontologies: we may think that the fruit ontology establishes which fruits are exotic and which are not. If the broker instance of the bdi1Broker class receives a request asking which agent sells passion fruits, it can look up the fruit ontology to discover that passion fruit is an exotic fruit, and thus it should be sold by the exoticSeller agent.

5. Development methodology

The CaseLP methodology we are going to discuss is general enough to be adopted for developing MAS prototypes or applications with any development toolkit: our methodology is not "CaseLP-dependent", even if CaseLP supports it with tools and languages, making it easier to follow. The product a software engineer will develop following this methodology is a *MAS prototype* and not a final application. Even if we use "MAS development", "MAS implementation" and "development of a MAS prototype" interchangeably, only the third sentence really reflects the target of CaseLP. We should remember, however, that this prototype can integrate external packages and legacy software: the more existing components are integrated into the prototype, the closer to the real application it becomes.

Developing a MAS prototype using the CaseLP methodology is characterized by the four phases described in Table I. The most complex phase is the *modeling* one since it is necessary to model the domain independent components, to chose which ones are most suitable for the current application

Table I. CaseLP methodology.

Phase 1: Modeling

- Domain independent components modeling
  - Architecture modeling
  - Role modeling
  - Ontology modeling
- MAS static structure modeling
  - Choice of the roles to be played
  - Understanding of the complete standard role model
  - Choice of the necessary ontologies
  - Grouping of roles in agent classes
  - For each class, choice of the most suitable architecture
  - Choice of the necessary initial agent instances
- MAS dynamics modeling
  - Definition of the program for each agent class
  - Definition of the initial state for each agent instance

Phase 2: Verification and validation

Phase 3: Implementation

Phase 4: Execution

and how they are combined to determine the static structure of the MAS. It is finally necessary to model the domain dependent components which determine the dynamics of the MAS.

5.1. MODELING

5.1.1. *Domain independent components modeling*

In this stage of the methodology, the domain-independent components are modeled. This stage can be skipped if all the domain-independent components necessary for the application have already been modeled: these existing models (specifications) can be reused for the current application.

5.1.2. *MAS static structure modeling*

In this stage, the pre-existing bricks (architectures, roles, ontologies) are composed together to provide a static description of the MAS to be developed. The task of the MAS developer is to choose "the right domain independent components to do the right thing". The steps to follow to achieve this goal are:

1. determine the roles to be played within the MAS;
2. determine the complete standard role model for the MAS, namely the interaction protocols which can or must take place among agents playing the roles identified in the previous step, in order to allow these agents to provide/require the services specified by the roles they play;
3. determine the necessary ontologies within the MAS;
4. group the roles in agent classes;
5. for each class, determine the most suitable architecture;
6. determine the necessary initial agent instances.

Some researchers in the agent community have provided suggestions for facing the steps above. Instead of developing new recipes for this purpose, we have adopted the existing suggestions and we have adapted them to the CaseLP methodology and framework. These informal recipes are collected in Appendix B of [64].

5.1.3. *MAS dynamics modeling*

In this modeling stage all the domain dependent behavioral decisions are taken and the specification of how the agents behave to fulfill their roles is given. In particular, the program characterizing the agent classes necessary for the application, as resulting from the previous step, and the initial state of the different agent instances must be provided.

5.2. VERIFICATION AND VALIDATION STAGE

If some specification languages used in the previous stages are executable, or if they support some kind of formal proof mechanism, verification and validation of properties are possible at this early specification stage. For example, this could be the case if role models are described using a formalism that supports deadlock detection mechanisms and the analysis of other interesting properties. The MAS developer will have great benefits from discovering bugs in the chosen role model before implementing it, or by being sure of the correctness of the protocols she will use thanks to a formal proof.

5.3. IMPLEMENTATION STAGE

After the MAS has been completely specified, either in its static aspects and in the dynamic ones, the developer must implement a working prototype. The implementation must be coherent with the specification given in the previous steps. Ensuring this coherence is demanded to the prototype developer, but semi-automatic translators can help her in implementing a prototype which satisfies the given specifications.

During the implementation stage, an aspect that we have neglected so far comes into play: the *integration of external software*. We delayed the treatment of external software integration to this point because it seems quite reasonable to have a MAS model that does not depend on the way services are provided. In the previous steps of the MAS modeling, we can *simulate* the functionalities provided by external software, without needing to know their details. This approach has the advantage of avoiding any commitment to very concrete choices until they need to be addressed in order to develop the prototype. At that time, the developer will have knowledge enough to take more motivated decisions. To integrate external software we comply with the FIPA *Wrapper Agent* model. As stated in [42], a wrapper agent allows other agents to dynamically connect to a software system uniquely identified by a software description. Client agents can rely commands to the wrapper agent and have them invoked on the underlying software system. The wrapper agent translates the commands contained in the requests it receives into operations on the underlying software, and translates the results back to the clients.

5.4. Execution stage

Once the prototype has been implemented, it must be executed to test all the choices made during the development stages. According to the resulting behavior of the prototype, some of these choices can be modified or refined. The modification/refinement may involve decisions taken in the early modeling stage as well as in the last ones, for example the choices related to integration of external software. The resulting modified prototype will be executed again and eventually modified/refined until it will show the expected behavior.

6. Tools and languages provided by CaseLP

CaseLP provides tools and languages for supporting each of the method steps described above. Some languages are suitable for more than one development step, as shown in Table II. Since we have widely discussed these tools and languages and the relations among them in [94, 64], here we just provide a short description of all of them. In Section 7 we will show the way some of them can be used in practice by means of the fruit marketplace example. We start describing the languages which can be adopted in more than one step of the methodology, namely HEMASL, an extension of UML and $\mathcal{E}_{hhf}$, and then we will describe the other ones.

Table II. Tools and languages supported by the CaseLP methodology.

Phase 1: Modeling

- Domain independent components modeling
  - Architecture modeling:
    HEMASL; UML extended; $\mathcal{E}_{hhf}$.
  - Roles and protocols modeling:
    UML extended; CaseLP-Protocol.
  - Ontology modeling:
    AgentOntology; CaseLP-Ontology; UML extended.
- MAS static structure modeling:
  CaseLP-MASadl, UML extended.
- MAS dynamics modeling
  - Definition of the program for each agent class:
    HEMASL; AgentRules; $\mathcal{E}_{hhf}$.
  - Definition of the initial state for each agent instance:
    HEMASL; AgentRules; $\mathcal{E}_{hhf}$.

Phase 2: Verification and validation:
The $\mathcal{E}_{hhf}$ interpreter.

Phase 3: Implementation:
ProlAg.

Phase 4: Execution:
CaseLP simulator; CaseLP visualizer.

6.1. HEMASL

CaseLP was born as a purely logic-based specification environment and all the languages it initially provided for specifying agent architectures, classes and instances were logic-based ones. There are many advantages in adopting logic-based languages for defining and executing MAS specifications, as discussed in Section 2 of [92], but there are also some limitations, as pointed out in Section 3.4 of [18]. For this reason we added a procedural language (HEMASL) and a graphical language (an extension of UML) to the set of languages offered by CaseLP. HEMASL [54, 56, 55, 94, 25] is a procedural meta-language for the specification of both agent architectures and agent programs and states. It is conceived as a language for designing and clarifying the abstraction hierarchy discussed in Section 4. Its most important features are:

Agent model. HEMASL supports the agent model we have introduced in Section 3.2. Since this model is an abstraction of many existing architectures, the development of heterogeneous architecture specifications is facilitated.

Hierarchy of abstraction levels. HEMASL provides constructs for specifying abstract and concrete architectures, agent classes and agent instances. This helps 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 in perceiving the events which take place in the environment and in performing actions which modify it. HEMASL provides an explicit model of the environment as well as primitives that can be used by agents to sense and modify it. It also provides constructs for modeling the message exchange among agents.

Operational semantics. The operational semantics of HEMASL is based on the concepts of "MAS configuration" and "configuration transition". It defines an abstract interpreter for the language[4].

6.2. UML

The current trends of research in the agent-oriented software engineering field put in evidence the advantages in adapting concepts, notations and methods associated with object-oriented analysis and design (and UML [87] in particular) to MAS analysis and design, such methodologies being increasingly familiar to a mass audience of software engineers. For this reason we designed a simple extension of UML to describe agent architectures and protocols complying with our CaseLP model [25]. This extension, based both on the Zeus methodology for agent specification [19] and on B. Bauer, J. Odell and H. V. D. Parunak's work [75], involves role diagrams, agent diagrams and protocol diagrams.

A *role diagram* specifies which roles are played by the agents in a MAS, how provided services are linked to required services and vice-versa and which classes "implement" which roles.

The *agent diagram* specifies the hierarchy of abstraction levels discussed in Section 4 by identifying the architectures of the agents in the MAS, the roles they fulfill and the relationships among them.

As far as *protocol diagrams* are concerned, they are a simplification of J. Odell's proposal [74]. Protocols are mapped into a diagram which is structurally similar to an sequence diagram with *roles* instead of general classifiers.

[4] This abstract interpreter has been designed and its implementation is under way.

Besides the diagrams we have just introduced, traditional UML *class diagrams* can be used to model ontologies and the static structure of the MAS and *behavior diagrams* can be used to express the agent's program.

6.3. $\mathcal{E}_{hhf}$

$\mathcal{E}_{hhf}$ [30, 31] is a linear logic language based on Forum [68] that can be used to specify concrete agent architectures, as well as their program and state. Specifications in $\mathcal{E}_{hhf}$ are executable, so it is possible to directly interpret the given specification. The language $\mathcal{E}_{hhf}$ can be used to well characterize an agent architecture under a semantic point of view. In fact, linear logic naturally models concurrent and state-based systems as agent are. Moreover it is useful for proving properties of protocols, as shown in [8]. In [3] an $\mathcal{E}_{hhf}$-based framework for MAS is defined. It is used to model MAS in which agents with heterogeneous architectures can co-exist.

6.4. Languages for ontologies

An almost natural way of modeling an ontology is as a 5-tuple

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

where $name$ identifies the ontology, $terms$ is a set of 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. This representation allows to define both domain independent ontologies and domain-dependent ones. The usage of this representation language named AgentOntology in the CaseLP environment has been widely discussed in [95], where an extensive example of use coming from the sport field is shown, and in Chapter 6 of [94], where the KICKER application discussed in Section 8.5 is described.

Another way to represent ontologies in CaseLP is by means of CaseLP-Ontology [25]. Also in this case an ontology is characterized by its name and the concepts belonging to the application domain, defined by means of a list of attributes.

6.5. CaseLP-MASadl

CaseLP-MASadl is a simple *architectural description language* for MAS that is adopted in CaseLP for describing the MAS overall structure in terms of agent roles, architectures, ontologies, classes, instances and their communication links.

In particular it is used to describe

- The roles necessary for the application.
- The satisfaction links among required and provided services.

- The hierarchy of roles, defined by means of containment and refinement relations.
- The agent classes, their subclasses, their architectures and the ontologies they understand.
- The initial agent instances.

Examples of use of CaseLP-MASadl can be found in [2] which describes the vehicle monitoring application summarized in Section 8.6, in Chapter 6 of [94] and in Appendix F of [25], where the static structure of the fruit marketplace example is fully specified.

6.6. AGENTRULES

AgentRules is a rule-based agent programming language used to define classes and instances of agents with a reactive–proactive architecture. It is very close to ProlAg, the language used in CaseLP for prototype implementation (see the next section) and just adds some syntactic sugar to it to provide prototype developers with a more user-friendly language. An agent is given an *initial state* that is a (possibly empty) set of atomic ground formulas. The initial state expresses the initial agent's knowledge about its application domain. An agent program is a set of reactive and/or proactive rules that define its procedural knowledge. The program of a reactive agent is expressed by means of *event-condition-actions* rules while proactive agents behave accordingly to *condition-actions* rules. AgentRules has been used to specify the behavior of agents in the KICKER application [94] and in the vehicle monitoring application [2].

6.7. PROLAG

Once all the agents in the prototype have been specified using the languages and tools described above, an executable prototype must be implemented to run a simulation. In particular, for each agent in the system, we must implement its initial state, its program and its engine. 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 languages, we have developed a platform-independent logical language for the implementation of the agents that are included in a prototype. ProlAg (*Prolog for Agents* [62]) is the target language for the implementation of the MAS prototype. It extends standard Prolog with primitives for *safe* state update and for message exchange. Two compilers exist for automatically translating ProlAg programs into ECLiPSE and SICStus Prolog. The external packages which can be accessed from the prototype are those ones for which ECLiPSE and SICStus

Prolog already provide interfaces, namely the ECLiPSE database and Tcl/Tk for ECLiPSE and the Java programming language, by means of the Jasper interface, for SICStus Prolog.

6.8. CASELP SIMULATOR: THE VISUALIZER AND THE SHELL

The execution of the prototype is based on a centralized *round-robin scheduler* that activates in turn all the agents in the MAS. When activated, an agent executes the cycle that implements its engine and behaves accordingly to the rules defining its program. When the agent terminates the execution of its cycle, the scheduler activates the next agent. The CaseLP scheduler controls the global clock of the MAS execution: once all agents have been activated in succession, the simulation time is increased by one, and another round starts; this process goes on until the established number of iterations has been reached. The CaseLP scheduler also manages the delivery of messages: when the current executing agent sends a message, the scheduler puts in into the mailbox of the receiver agent eventually adding a random delay in a range established by the developer during the configuration of the simulation run. Using the CaseLP Shell (see [2] for an example), which allows to type in commands to execute the simulation, it is possible to establish the failure rate and minimum and maximum latency for each communication channel, the number of execution cycles, the initial messages for each agent. Once the execution is terminated, it is possible to assign a weight to each message exchanged during the simulation run and collect statistics such as the number of messages received in any simulation interval, the sum of their weights, the total number and weight of exchanged messages, the maximum and average number of messages per interval, etc.

The CaseLP Visualizer [78] provides the software engineer with a user-friendly GUI to load the agents into the system, start the execution of the prototype, and trace the events that take place during simulation. It allows both on-line and off-line visualization of the MAS execution, tracing agent state updates and message exchange and showing statistics about sent messages.

7. The distributed marketplace example

7.1. DOMAIN INDEPENDENT COMPONENTS MODELING

To model the distributed marketplace example depicted in Figure 3 following the CaseLP methodology we need to be aware of the available domain independent components, namely architectures, role models and protocols and domain independent ontologies. We may take advantage of libraries of domain independent components specified using languages supported by CaseLP as well as libraries of components specified using other languages.

For example, the HEMASL specifications of the bdi abstract architecture, of the bdi1 and bdi2 concrete architectures and of a simple reactive architecture (both abstract and concrete) are available [25], as well as the $\mathcal{E}_{hhf}$ specification of the bdi1 concrete architecture [3]. CaseLP provides an agent ontology for application independent concepts about agents, their architectures, their roles and their services (Chapter 4 of [94][5]), and a domain independent ontology for a distributed marketplace application (Chapter 8 and Appendix G of [25]). Finally, as far as roles and protocols are concerned, even if in CaseLP we developed no library yet, we can refer to libraries of role models that already exist (see [85, 19] and the FIPA interaction protocols specifications, `http://www.fipa.org/repository/ips.php3`).

7.1.1. *HEMASL specification of the BDI architecture*

In the piece of code shown in Table III the bdi abstract architecture is defined (line abs-1) and in the code shown in Table IV it is then specialized into the bdi2 concrete architecture (line conc-1).

In the abstract architectured definition, type variables are declared first: they will be substituted with the proper type definition when the concrete architecture will be defined (see lines abs-2 to abs-6 and conc-2 to conc-8). Then, the architecture abstract components are declared. For each component, the developer specifies if it is a program component, namely, it characterizes the class of the agent, or if it is a state component characterizing the instance of an agent. In lines abs-9 and abs-10 the `plans-component` is declared as a structure (not yet specified) containing objects of type `Plan`, and characterizing the agent program (note the `class` keyword before the declaration). In lines conc-10 to conc-15 the abstract components are assigned a type; for example `plans-component` is defined as a `list`. The next step is to define the abstract procedures (lines abs-17 to abs-23), namely those procedures whose implementation will be provided in the concrete architecture (lines conc-17 to conc-26). The procedures that will be used in the engine are then defined in terms of the abstract procedures (lines abs-25 to abs-29) and finally the specification of the engine is provided (lines abs-31 to abs-40). There is no need to refine the engine cycle in the concrete architecture: this cycle will turn out to be implemented when the abstract procedures it involves will be implemented.

7.1.2. $\mathcal{E}_{hhf}$ *specification of the bdi1 concrete architecture*

In table V we show a piece of code implementing the bdi1 concrete architecture in $\mathcal{E}_{hhf}$. Informally speaking, given a rule *Condition* $\Rightarrow$ *Head* $\multimap$ *Body* the formulas at the left of the $\Rightarrow$ connective represent a condition to be tested in order for the rule *Head* $\multimap$ *Body* to fire (variables are universally

[5] Our conceptualization of agents has slightly changed in the meanwhile and this ontology should be revised according to our new agent model.

Table III. Specification of the bdi abstract architectures in HEMASL.

```
abstract_architecture {bdi} {
type_declaration {
  varType Plan;
  varType Belief;
  varType Intention;
  ............};

components {
    class plans_component:
                             component(Plan);
   agent beliefs_component:
                           component(Belief);
   agent intentions_component:
                        component(Intention);
   ............};

 abstract_procedures{
    get_action(intention : Intention,
                          !action : Action);
    get_event_trigger(!trigger : Trigger);
    get_plan_trigger(plan : Plan,
                        !trigger : Trigger);
    ............};

  procedures {
    percept_event() {
      percept_event_from_extern()
    };
    plan_triggering() {
      ............};

  engine {
  \* declarations of local variables *\

    while (true) do {
      percept_event();
      plan_triggering();
      plan_selection();
      upgrade_intentions_component();
      exec_intention()
    } endwhile }
  };
```

Table IV. Specification of the bdi2 concrete architectures in HEMASL.

```
architecture {bdi2} is a {bdi} {
 type_definition{
   varType Plan: tupla(3,Trigger,
                     Conditions, Intention);
   varType Belief: tupla(2,BelIde,
                                 BelValue);
   varType Intention: list(Action);
   ............};

 components {
   plans_component: list(Plan);
   beliefs_component: list(Belief);
   intentions_component: set(Intention);
   triggered_plans: list(Plan);
   ...........};

 procedures {
   percept_event_from_extern() {
     decl mex: Message;
     receive(All, mex);
     events.insqueue(mex);
   }
   get_event_trigger(!trigger : Trigger){
     trigger := selected_event.gettupla(1)
   };
 };
}
```

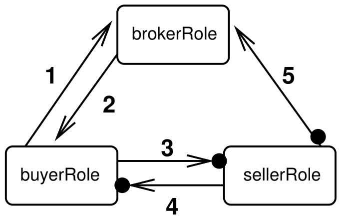
quantified). When a rule fires, the formulas in the head are *consumed* and the formulas in the body are *produced* and become available in the new state.

The following clause belongs to the set of rules defining the behavior of a bdi1 concrete architecture. It formalizes what to do to manage an incoming message of the form *event(Trigger, Id)*, with the trigger being of type internal: if there is a plan in the agent's knowledge base whose triggering event matches *Trigger*[6] and *Trigger* is internal, the intention stack with the same identifier as the triggering event is updated by adding a new plan instance triggered by *Trigger* on top of it, provided that the condition *Cond* holds. In

[6] Since $\mathcal{E}_{hhf}$ is a logic-based language, we use unification [86] to match patterns.

Table V. Specification of a concrete architecture in $\mathcal{E}_{hhf}$.

$$
\begin{array}{l}
\textit{plan\_library(List\_of\_plans)} \mathbin{\&} \textit{internal(Trigger)} \mathbin{\&} \\
\textit{member(plan(Trigger, Cond, Body), List\_of\_plans)} \\
\quad \Rightarrow \textit{plan\_triggering(event(Trigger, Id))} \invamp \\
\quad \textit{intention\_stack(Id, [plan\_inst(}B_1\textit{, }S_1\textit{)} \mid \textit{Tail])} \multimapinv \\
\qquad \textit{verify(Cond)} \mathbin{\&} \\
\qquad \textit{intention\_stack(Id, [plan\_inst(Body, active)}\mid \\
\qquad \textit{[plan\_inst(}B_1\textit{,}S_1\textit{)} \mid \textit{Tail]]))}.
\end{array}
$$



	Collaboration	Explanation
1	Find Request	Asks for agents selling items of interest
2	Find Response	A list of agents matching the desired criteria
3	Buyer offer	Message containing buyer's proposal
4	Seller response	Seller's reply to a previously submitted offer
5	Register	Sellers register or de-register their items for sale

Figure 7. Marketplace role model and initial interactions.

[3] and [10] the complete set of clauses for managing the incoming events of different types is defined.

7.1.3. *Marketplace role model*

To specify the role model (namely, the interactions among roles) suitable for the distributed marketplace we refer to an already developed library of role models: the Zeus role modeling guide [19]. This marketplace role model is depicted in Figure 7. In this role model agents playing the brokerRole, sellerRole and buyerRole roles interact for reaching the best agreement to exchange goods.

7.1.4. *The contract proposal protocol*

In Figure 8 we show an application of UML extended (the "xor" connective represented by a diamond with an x inside is not standard) to specify protocols; agents playing the buyerRole role can send a contractProposal

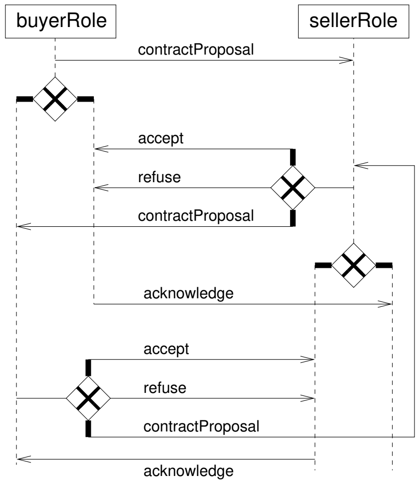


Figure 8. The contract proposal protocol in UML.

message to agents playing the sellerRole role, who can accept, refuse or make a counter proposal to the buyer. The interactions take place among roles and not among instances of agents; this makes the protocol as general and reusable as possible.

7.1.5. *The marketplace ontology*

We organize concepts about the distributed marketplace application by defining an ontology where we establish that, when a contract is proposed by an agent to another agent, this contract must specify the name of the proposer (for example, nonExoticBuyer), the name of the receiver (for example, nonExoticSeller) and the merchandise to be exchanged . The merchandise is in turn characterized by a name (for example, apple), a positive cost the proposer is willing to pay and a positive quantity it is willing to buy. A contract ends when one participating agent accepts or refuses the contract and the other agent sends an acknowledge, according to the contract proposal protocol shown in Figure 8. This ontology can be easily expressed in CaseLP-Ontology (Table VI) and in UML (Figure 9).

7.2. MAS STATIC STRUCTURE MODELING

In our simple marketplace application, it is quite easy to identify the roles to be played: the fruit marketplace is characterized by exotic and non-exotic fruit buyers, exotic and non-exotic fruit sellers and a marketplace broker which tries to accommodate requests and offers. Three general roles are thus necessary:

– buyerRole,

Table VI. Specification of the marketplace ontology in CaseLP-Ontology.

```
ontology marketplace{
    Merchandise{
      name: string;
      cost: int [cost>0];
      quantity: int [quantity>0];
    }
    Contract{
       proposer: string [Contract.proposer
                                   = AgentName];
       receiver: string [Contract.receiver
                                   = AgentName];
       product: Merchandise;
    }
    ContractEnd{
      reason: string [reason = "acknowledge"]
    }
   }
```

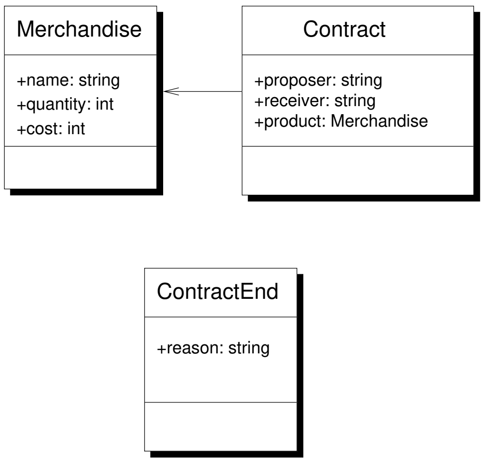


Figure 9. Specification of the marketplace ontology in UML.

- sellerRole,
- brokerRole.

These roles must respect the role model and the interaction protocol described in the previous section.

As far as the necessary ontologies are concerned, we can use the marketplace one described in the previous section. We should also define a fruit ontology, specifying which fruit is an exotic one and which is not. Since our running example is fairly simple, we do not need more complicated ontologies.

Now that we have clear in mind the roles played by the agents and the main concepts involved in our application, we can group roles in agent classes and assign the most suitable achitecture to each class. The way the grouping is performed is shown in Figure 10, where classes are represented by rectangles with sharp edges and roles are represented by rectangles with rounded edges. Each role is characterized by its name shown in the upper compartment, by the services it requires, listed in the middle compartment, and by the services it provides, listed in the bottom compartment. The links between provided and required services are represented by arrows with the name of the service above them. For example, the sellerRole role requires the register_sellers service which is provided by the brokerRole role. Every agent playing the sellerRole role or the buyerRole role interacts only with one agent playing the brokerRole role, while the vice-versa does not hold (note the multiplicity of the locate_sellers and register_sellers links). Classes are identified by their name: in the diagram there are one bdi2Buyer class playing the buyerRole role, one reactProact1Seller class playing the sellerRole role and one bdi1Broker class playing the brokerRole role. The reactProact1Trader class is the most interesting one, since it shows how a class can play more roles. In fact, instances of this class will play both the sellerRole and the buyerRole roles. This class was not shown in Figure 3 for sake of readibility.

Even if the names assigned to agent classes suggest which architecture they will have, the role diagram depicted in Figure 10 does not explicitly specify the architecture associated to them. This association is determined by the agent diagram shown in Figure 11. The bdi1Broker class is derived by the bdi1 concrete architecture by filling its program components with the necessary knowledge to operate in the marketplace domain and to play the brokerRole role. In a similar way, the bdi2Buyer class has a bdi2 concrete architecture and the reactProact1Seller and reactProact1Trader classes have a reactProact1 concrete architecture.

Finally, to summarize the complete static architectural model for our marketplace application including the necessary agent instances and the ontologies understood by the agents, we can use CaseLP-MASadl, as shown in Table VII. The table also shows that CaseLP-MASadl can be used to define the way provided and required services can be linked, and which conversation should be followed to provide or require one service. The conversation locate

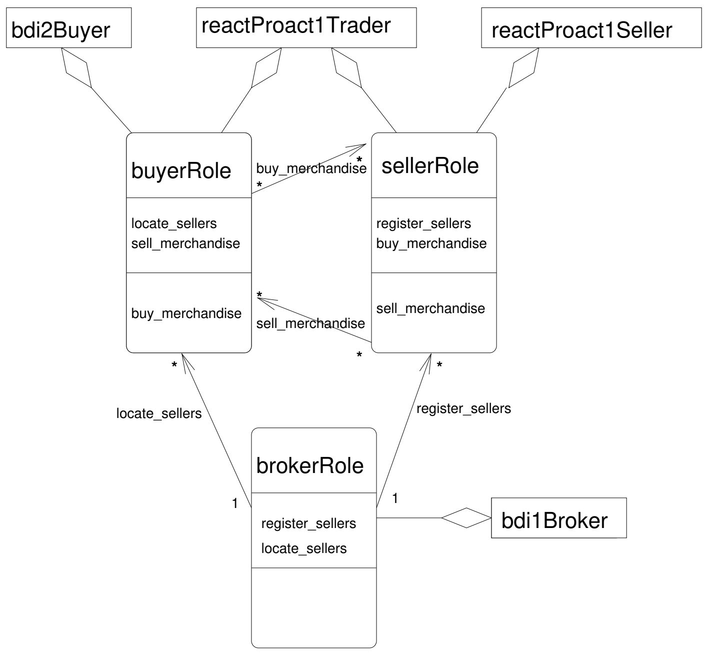


Figure 10. The role diagram for the marketplace application specified in UML.

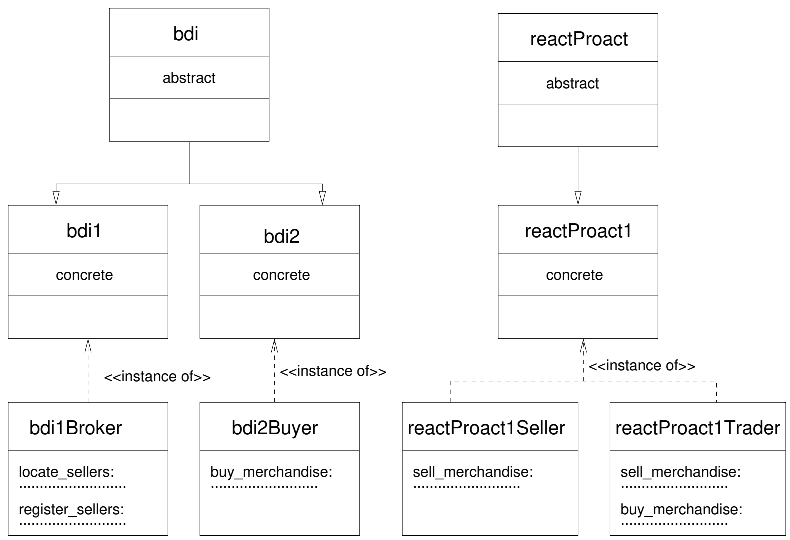


Figure 11. The agent diagram for the marketplace application specified in UML.

that buyers should follow to locate the sellers of a specific product is shown as an example.

7.3. MAS DYNAMIC MODELING

This stage involves the definition of the program of each class involved in the application and the state of each agent instance. The languages we can use in this stage are HEMASL, AgentRules and $\mathcal{E}_{hhf}$. In the following paragraphs we exemplify their use by showing a piece of code in each of them.

7.3.1. *HEMASL*

Table VIII shows the initialization of the class components of the bdi2Buyer class, which has a bdi2 architecture and plays the buyerRole. As shown in lines conc-3 and conc-9 of Table IV, plans are lists of tuples, where each tuple contains a trigger, a list of conditions and a list of actions representing the agent's intentions. When the amount of the product prod_name is below 25 Kilograms (triggering event), the buyer decides to contact the seller of this product and proposes a contract for buying the needed amount of merchandise (body of the plan), at a price that the buyer had previously established and stored in its state (conditions over the state). Analogous plans must be provided to cope with all the possible triggering events.

The instructions `init_ENV[]` and `init_ETH[]` are used to initialize the environment and ether components to the empty lists.

Finally, `create_agent(bdi2Buyer, nonExoticBuyer)` allows to define one instance of the bdi2Buyer class, namely the nonExoticBuyer, by initializing its state components.

7.3.2. *AgentRules*

The program and initial state of agents belonging to the reactProact1Seller class can be easily defined using AgentRules. In Table IX we show part of the initial state of the nonExoticSeller agent and the reactive rule which establishes the way it decides to reply to a contract proposal sent by a buyer agent.

7.3.3. $\mathcal{E}_{hhf}$

Finally, we can define the program components of the bdi1Broker class and the state components of its broker instance in $\mathcal{E}_{hhf}$. Table X shows a plan whose triggering event is the reception of a register request from a seller agent.

Table VII. Static MAS architecture and conversational model in CaseLP-MASadl.

```
role sellerRole {
  requires buy_merchandise; register_sellers
  provides sell_merchandise;}
role buyerRole {
  requires locate_sellers, sell_merchandise;
  provides buy_merchandise;}
role brokerRole {
  provides register_sellers, locate_sellers;}

agentclass reactProact1Seller {
  roles: sellerRole;
  architecture: reactProact1;
  ontologies: agent, fruit, marketplace;}
agentclass bdi1Broker {
  roles: brokerRole;
  architecture: bdi1;
  ontologies: agent, fruit, marketplace;}
..............................

agentInstance{
 broker            bdi1Broker;
 exoticBuyer, nonExotiBuyer     bdi2Buyer;
 exoticSeller, nonExoticSeller  reactProact1Seller;}

link{
sellerRole.buy_merchandise  <--
                      buyerRole.buy_merchandise;
sellerRole.register_sellers <--
                      brokerRole.register_sellers;

buyerRole.locate_sellers   <--
                      brokerRole.locate_sellers;
buyerRole.sell_merchandise <--
                      sellerRole.sell_merchandise;}

Conversation locate (buyerRole.locate_sellers <--
                      brokerRole.locate_sellers)
1) message: buyerRole --> brokerRole
   performative: ask
   content: find([Product_1, ..., Product_n])

2) message: brokerRole --> buyerRole
   performative: reply
   content: found([
              (Product_1, [Name_1_1, ...,Name_1_n]),
                   ...,
              (Product_n, [Name_n_1, ..., Name_n_m])])
```

Table VIII. Definition of the agent program and initial state in HEMASL.

```
class_agent {bdi2Buyer}{bdi2} roles{buyerRole} {
        init_comp{
          plans_component = [(
\* trigger *\     amount(prod_name)<25,
\* conditions *\
         [ (prod_name, buyer_price, avail_qnty) =
                       get(products, prod_name),
           (prod_name, seller_name) =
                       get(sellers, prod_name),
            my_agent_name = get(agent_name),
            buyer_desired_qnty = 25 - avail_qnty]
\* body *\        [send(contractProposal(
                      my_agent_name, seller_name,
                      (prod_name, buyer_price,
                         buyer_desired_qnty)),
                  my_agent_name)]), .......]; }; }

      init_ENV [];
      init_ETH [];

      create_agent(bdi2Buyer, nonExoticBuyer); {
        beliefs_component = [
          (agent_name, nonExoticBuyer),
          (products, [(apple, 1.50, 20),
                      (pear, 1.80, 15)
                      (peach, 2.50, 10),
                      (plum, 2.00, 12)]),
          (sellers,  [(apple, nonExoticSeller),
                       .....]), .....];
        intentions_component = [.....];
      }
```

7.4. Implementation

We skip the verification and validation phase and we go directly to the implementation one: executing the specification of the marketplace components is enough to obtain a feedback about the application behavior.

In order to implement the agent instances we have specified so far, it is necessary to translate their specifications into the implementation language of CaseLP prototypes, ProlAg. We do not need to integrate external packages,

Table IX. Definition of the agent program and initial state in AgentRules.

```
program
  product(apple, 180, 50)
  product(pear,  210, 100)
  product(peach, 240, 60)
  product(plum,  220, 80)
  ....................

  on message contractProposal(receiver, proposer,
                      (prod_name, buyer_prop_price,
                          buyer_required_quantity))
  check      product(prod_name, seller_price,
                          seller_available_quantity)
             and  buyer_prop_price + 0.1 *
                          buyer_prop_price >=
                                         seller_price
             and  buyer_required_quantity <=
                          seller_available_quantity
  do         send(accept(proposer, receiver,
                      (prod_name, buyer_prop_price,
                          buyer_required_quantity)))
```

Table X. Definition of a program component in $\mathcal{E}_{hhf}$.

```
(plan(
  \* Trigger *\      register(Name, [P_1, ..., P_n]),
  \* Conditions *\ trusted(Name),
  \* Body *\         assert_state(registered(Name, [P_1, ..., P_n])))
```

so we will use only ProlAg to implement the prototype. Due to their rule-based nature, AgentRules and $\mathcal{E}_{hhf}$ specifications can be easily translated into ProlAg rules, while HEMASL, mainly conceived as a language for designing and clarifying the conceptual structure of the architecture hierarchy, is not suitable for being translated into ProlAg code.

7.4.1. *From AgentRules to ProlAg*

The implementation of an AgentRules specification in ProlAg conforms the following schema.

```
fact(B_1).              .........           fact(B_n).

reactive_rule(Message_1, Condition_1, Actions_1).
```

```
reactive_rule(Message_m, Condition_m, Actions_m).

proactive_rule(Condition_m+1, Actions_m+1).
proactive_rule(Condition_m+k, Actions_m+k).

p_1(..):- .........
.......
```

The state of an agent is specified in AgentRules as a set of ground facts `B_1, ..., B_n`, that represent the agent's beliefs. It is immediate 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. An AgentRules 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 immediate to define the mapping from proactive AgentRules rules into ProlAg. The second set of clauses `p_1, ..., p_s` that are included in a ProlAg program corresponds to the set of AgentRules auxiliary procedures that are used to verify auxiliary conditions. The engine for a reactive-proactive agent can be implemented in ProlAg as shown below[7]:

```
engine :-
  async_receive_one(Msg_input, Msg_output),
      Msg_output == no_message ->
         ReactActions = [];
         select_reactive_rule(Msg_output,
                                ReactActions)),
  select_proactive_rule(ProactActions),
  choose(ReactActions, ProactActions, Actions),
  execute(Actions).
```

The agent inspects its mailbox and gets a message, unified with the `Msg_output` variable, which matches the pattern specified by the `Msg_input` variable. If there is no message in the mailbox matching this pattern, the set of

[7] The `(Condition -> Pred1 ; Pred2)` statement is an if-then-else statement.

reactive actions to perform is empty (`Msg_output == no_message -> ReactActions = []`), otherwise the actions to perform are found by selecting a reactive rule whose triggering message matches with `Msg_output` and whose associated condition is verified in the current state (`select_reactive_rule(Msg_output, ReactActions)`). In both cases, the successive step is to select a proactive rule whose condition is verified in the current state, and determine the actions to perform according to this rule (`select_proactive_rule(ProactActions)`). Finally, the reactive and proactive actions are combined together to find a set of actions to execute.

7.4.2. *From $\mathcal{E}_{hhf}$ to ProlAg*

In order to translate an $\mathcal{E}_{hhf}$ specification into a ProlAg program, we start with observing that the $\mathcal{E}_{hhf}$ clauses 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 in ProlAg, we introduce an auxiliary predicate `aux_C`, defined as

```
aux_C :-
   G_1, ..., G_m,
   retract_state(A_1), ...,retract_state(A_n),
   G_m1, ...,G_mk,
   assert_state(A_n1), ...,assert_state(A_nh).
```

where `retract_state` and `assert_state` are the safe state update predicates provided by the language. The proof of `G_1`, ..., `G_m` tests the clause guard, `retract_state(A_1)`, ..., `retract_state(A_n)` consume the atomic formulas `A_1`, ..., `A_n`, the proof of `G_m1`, ..., `G_mk` verifies the conditions over the current state, and the actions `assert_state(A_n1)`, ..., `assert_state(A_nh)` add new information to the state itself. By applying this transformation to every $\mathcal{E}_{hhf}$ clause, we can obtain the corresponding set of ProlAg procedures. Such a set can be partitioned by grouping together clauses regarding the five main activities performed by a BDI engine that are *event_perception*, *plan_triggering*, *plan_selection*, *plan_pushing* and *int_stack_selection_and_action*. The top-level behavior of a BDI interpreter can be given by the rule that starts the five main activities, ordered according to their priority[8].

```
engine :- event_perception;
          plan_triggering;
          plan_selection;
```

[8] The “;” operator is an or. The Prolog interpreter starts executing the predicates in an “or” formula from left to right.

```
        plan_pushing;
        int_stack_selection_and_action.
```

Obviously, more sophisticated strategies can be implemented, so that the execution of the five activities may take additional information into account.

Each macro-activity can be defined by grouping the corresponding clauses together:

`plan_triggering :-aux_plan_trigg_1; ... ;aux_plan_trigg_s.`

where `aux_plan_trigg_1; ... ;aux_plan_trigg_s` are the clauses of the plan triggering group. As an example, the $\mathcal{E}_{hhf}$ clause described in Section 7.1 will look like the following:

```
aux_plan_trigg_K :-
   /* resources to be consumed */
   retract_state(plan_triggering(event(Trigger,
                                          Id))),
   retract_state(intention_stack(Id,
              [plan_inst(B_1, M_1, S, SA)|L_1])),

   /* conditions to be verified */
   plan_library(List_of_plans),
   member(plan(Trigger, Context, Body, Maint,
                    SuccActs), List_of_plans),
   internal(Trigger),
   verify(Context),

   /* resources to be produced  */
   assert_state((intention_stack(Id,
     [plan_inst(Body, Maint, active, SuccActs)|
            [plan_inst(B_1, M_1, S, SA)|L_1]]))).
```

7.5. Execution

Once all the agents have been implemented in ProlAg, we can execute the MAS specification to verify that it behaves like expected. We discuss the execution carried out with the CaseLP Visualizer whose graphical interface simplifies the configuration and debugging of the application. The CaseLP Shell provides the same configuration functionalities and even more sophisticated functionalities to get statistical results on the simulation run.

The first window of the CaseLP Visualizer allows the developer to choose whether to execute a new simulation or to analyse the results of a previously executed one. If the user decides to run a new simulation, she can choose the simulation to run from a menu. Then she can choose which agents should be "instrumented" (Figure 12). The code of instrumented agents is automatically

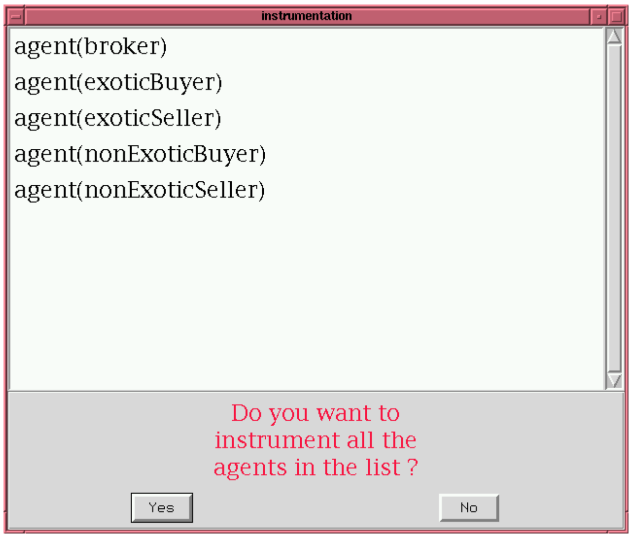


Figure 12. Choice of the agents to instrument.

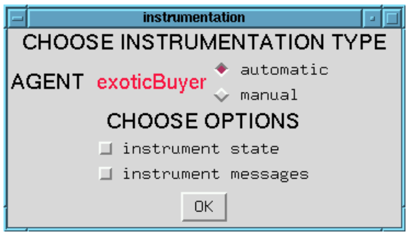


Figure 13. Choice of the instrumentation mode for each agent.

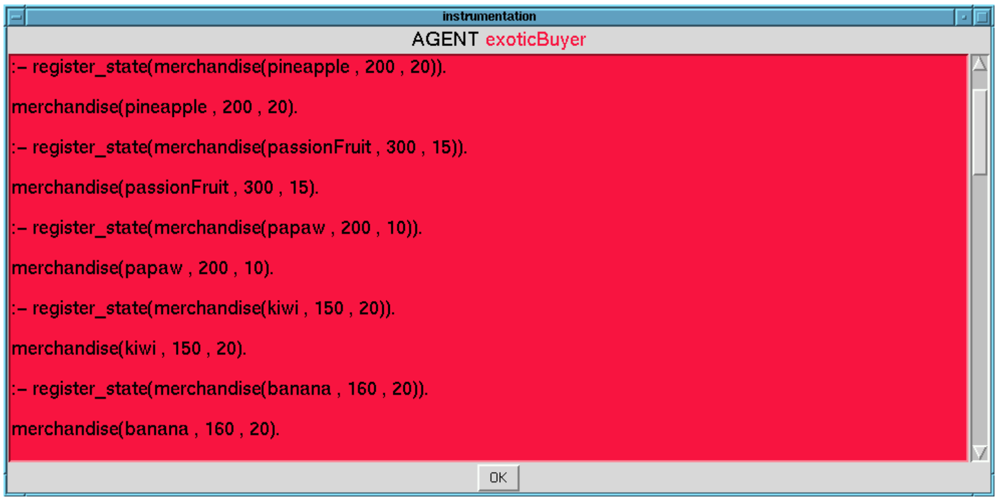


Figure 14. Instrumented code of the exoticBuyer agent.

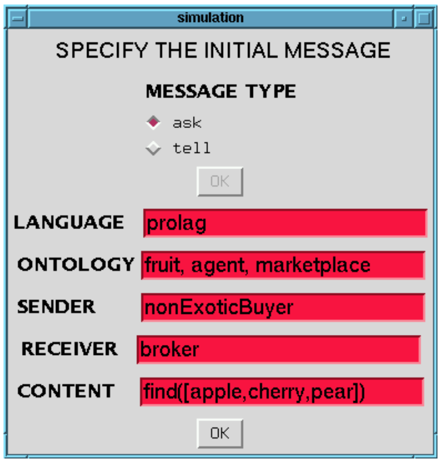


Figure 15. Specification of the initial messages.

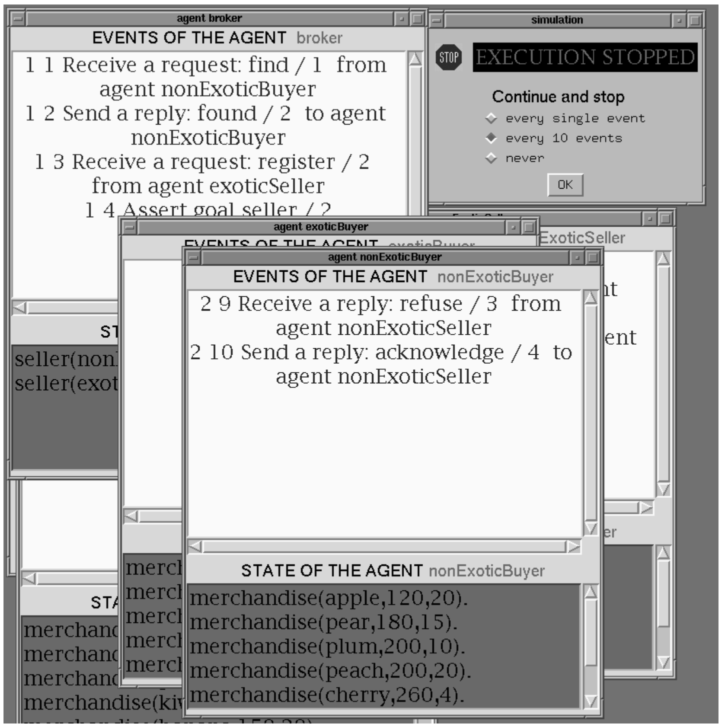


Figure 16. Online trace of the simulation evolution.

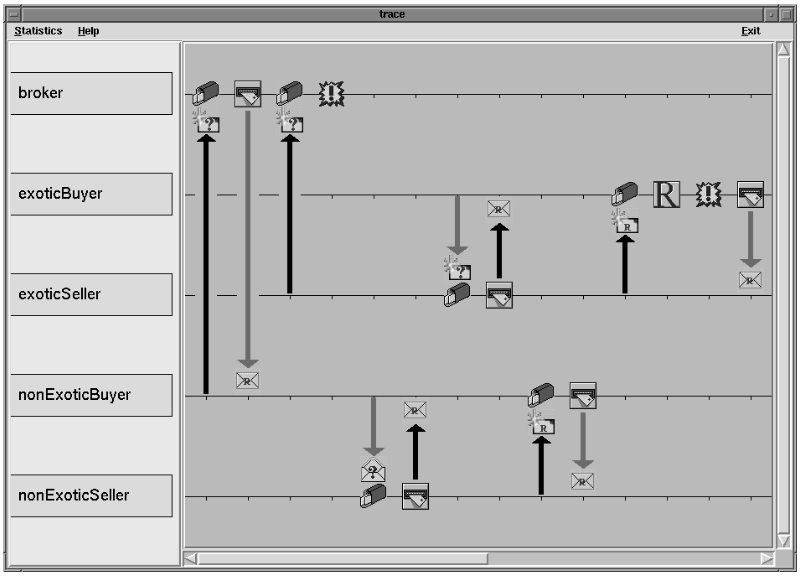


Figure 17. Off-line trace of the simulation evolution.

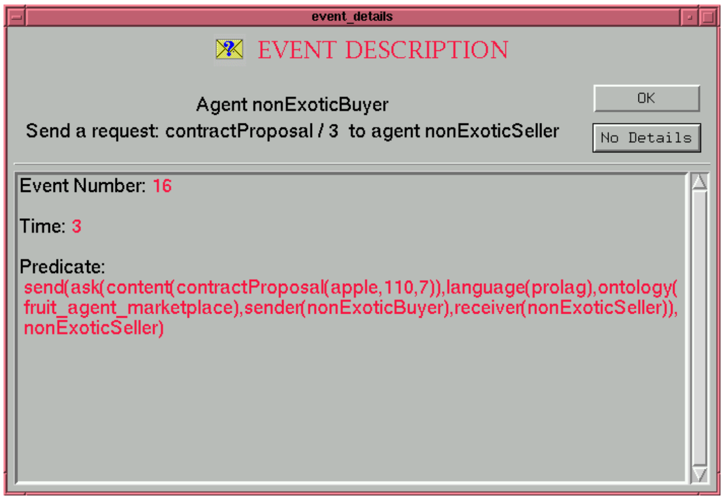


Figure 18. Details on exchanged messages.

extended with instructions for tracing the exchange of messages and/or state updates, according to the developer's preferences (Figure 13) and it is then shown to the developer who can manually modify it (Figure 14). Once the agents have been instrumented, the user can specify the initial messages they will receive (Figure 15), and then the simulation can start. The developer can follow the simulation evolution thanks to windows that show, for each instrumented agent, the messages it sent and received and the way its state

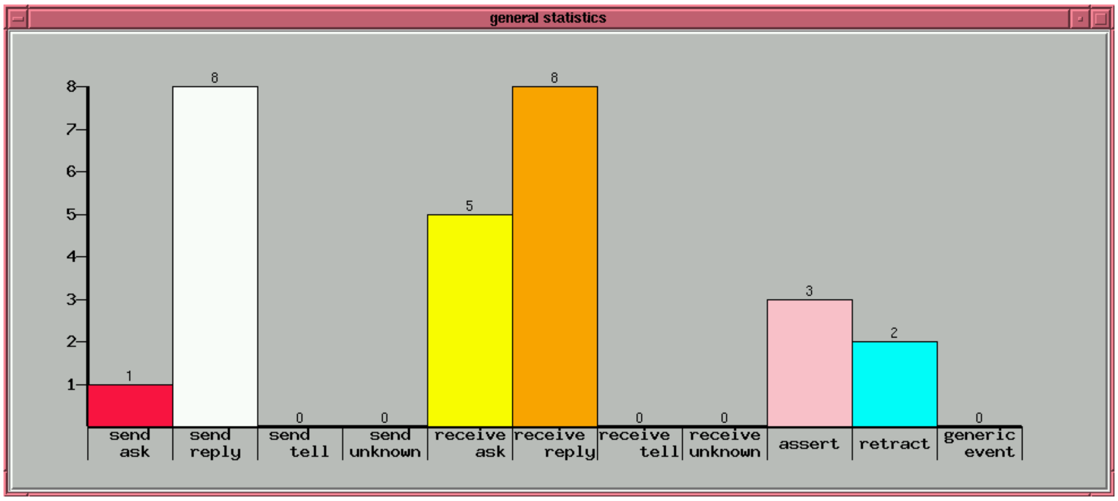


Figure 19. Statistics on the exchanged messages and state updates.

is updated (Figure 16). After this on-line check the developer can perform an off-line analysis of the overall simulation execution (Figure 17) thanks to a windows which graphically represents the message flow among the agent instances. The window for off-line trace allows to inspect details of message exchange (Figure 18) and state updates (by clicking on the desired icon) , and provides the means to obtain statistics on the number of exchanged messages and the types of state updates (Figure 19).

8. Applications of CaseLP

Since the beginning of the project CaseLP has been used for modeling and prototyping a large number of applications, many of them in collaboration with industrial partners. In this section we describe some of them.

8.1. Goods transportation

This application has been realized in cooperation with Elsag s.p.a., an Italian company that operates on international scale providing services automation. It was developed using the earliest version of CaseLP and is fully described in [63].

Four classes of agents are involved in this application: client, agency, distributor and transporter, as exemplified in Figure 20 where one instance of the client class, one instance of the agency class, six distributors and two transporters are present. A client asks to the agency to find a set of solutions for sending goods from one place to another one. The agency knows which distributors are present in the starting place and contacts them to make a

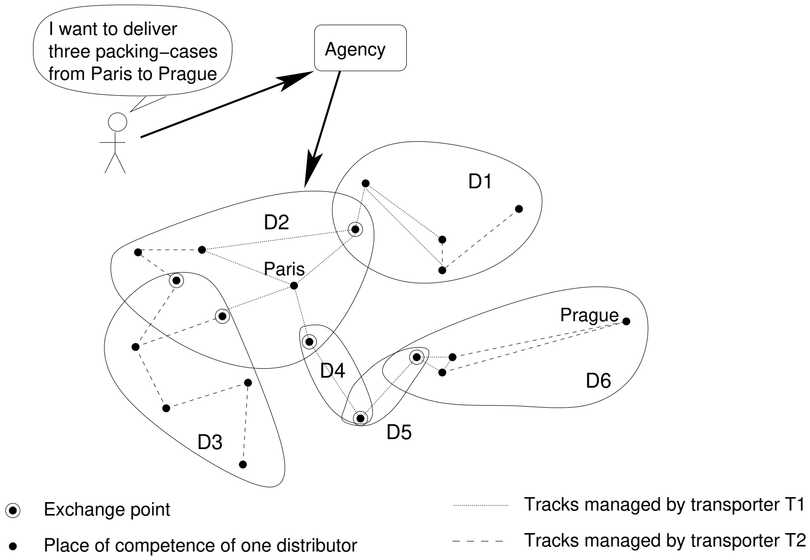


Figure 20. An example of planning goods transportation application.

plan for delivering the goods. Each distributor covers a limited portion of the territory and has no global knowledge of the way the other distributors are organized and which part of the territory they cover: it can only take the goods to exchange points where other distributors are present and can take them. When one distributor is contacted by another distributor to go on planning the delivery of goods, it recursively applies the algorithm by contacting other neighboring distributors. The process ends when one distributor discovers that the destination is within its territory or when it is discovered that the destination is not connected to the starting place. The planning phase is followed by a delivering phase, when goods are delivered along the path chosen by the client among the found ones. The transporters carry out the physical transport.

This application was the first one developed with CaseLP: it allowed us to identify some restrictive choices and limitations of the environment which have been overcome in successive versions; it also demonstrated the usefulness of an environment like CaseLP to our industrial partners: despite to its complexity, the working prototype of this application was developed and debugged in less than two person-months.

8.2. Distributed health care

Clinical data about one patient are usually spread across different locations for many reasons (hospitalizations in different structures, or in different wards inside the same structure); the administrative information about the patient, like its medical identifier, insurance, right to payment exemption, etc, are maintained in still different places. Even if we assume that all the information is maintained in a digital format, which is not always the case, finding the necessary information may be a very hard task.

In [80] a prototype of a MAS for facing this problem is described. The prototype has been developed in collaboration with the Knowledge and Data Engineering Group of the Trinity College, Dublin (Ireland), using CORBA [91] as the middleware for the distribution of the system. The adoption of CORBA allowed to easily implement the distribution, but suffered from two main limitations:

- the synchronous exchange of messages, which blocks the message sender until the receiver provides an answer; and

- the need to identify the set of necessary messages during the design phase, in order to use static invocation for calling operations on objects (dynamic invocation was too complex to be implemented in the prototype).

After being developed with CORBA, the prototype of the MAS was specified with CaseLP. The purpose of this reverse engineering process was to compare the features of message exchange with CORBA, as resulting from the experiments with the CORBA prototype, and message exchange with a high-level agent-communication language like KQML [66] or FIPA-ACL [41], used in the CaseLP simulation. This comparison confirmed that CORBA is a powerful tool for message physical transport, but not a flexible agent communication language. Asynchronous message exchange supported by high-level agent communication languages is definitely a better choice when autonomous, intelligent agents are involved in the application.

8.3. Mail video-encoding and sorting

Another application developed in collaboration with Elsag s.p.a. involved the analysis of a system for mail sorting which included on-line and off-line video-encoding processes [21].

In Italy, the video-encoding process takes place in the same centers where mail pieces arrive and are sorted. These postal centers are uniformly distributed over the territory, but the mail flow is concentrated in the Center and North of Italy. This means that personnel working at video-encoding in the Center and North of Italy has more work to face than the staff in the South of Italy. Elsag s.p.a. was interested in figuring out how the work could be redistributed if a network was used to send images of mail pieces received in one postal center in the North or Center to another postal center in the South, to be video-encoded remotely.

CaseLP was used to build a model of the existing postal system enriched with this network and to run a large set of simulations; the number of postal centers, their interconnections, the number of optical character recognition devices available in each center as well as the number of persons working at video-encoding in each center were parameterizable. The simulation runs could be configured to find the best organization of a network of postal centers given a fixed amount of resources (either staff or physical devices), the minimum resources required to have mail pieces sorted within a fixed amount of time, and so on. Our industrial partners recognized that CaseLP was more flexible than the ad-hoc simulators they were accustomed to, like WITNESS (`http://www.witness-for-simulation.com/`). On the other hand, the lack of user-friendliness of CaseLP was recognized to be its main limitation for an adoption in an industrial setting. According to this consideration, we started working at the CaseLP Visualizer described in Section 6.8 which enriches CaseLP with the user-friendliness it needed.

8.4. Freight trains traffic

This application has been developed within the framework of the EuROPE-TRIS Project [76] as a result of an industrial collaboration with the Information Systems Division of Italian Railways (FS s.p.a.) and is fully described in [22, 23].

It deals with the so-called train dispatching problem concerning the control of trains movement over a railway line, in order to guarantee traffic regularity and service performance. Train dispatching is subject to several operational constraints, particularly when deviations from scheduled timetable and other running disturbances (e.g. stochastic delays, unforeseen track maintenance, etc) occur. The goal of this application was to conceive a preliminary support system which could help the rail dispatchers or controllers in taking the best decisions and regulating actions (e.g. overtakings and temporary train stops at appropriate stations).

The real case-study we choose to demonstrate the suitability of the MAS technology and of CaseLP in particular to this kind of applications was the railway line connecting the Italian stations of Milano and La Spezia. The demonstrator we developed was used as the base for a decision support system for *Train Dispatching* to be supplied to the *Traffic Coordinator* in Milano (COIM) and the *Production Deputy* in La Spezia (yard operative manager). The Italian Railways partners were satisfied with the results obtained during this collaboration and we were asked to extend this first prototype to cope with more complex scenarios on a longer railway line.

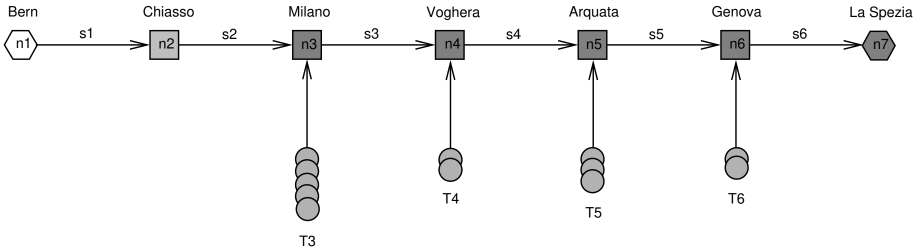


Figure 21. Railway line connecting Bern to La Spezia.

8.5. KICKER

KICKER [94] is the evolution of the project depicted above. It is based on the module F-TTM of the European project EuROPE-TRIS that realized a set of demonstration tools that provide traffic operators with several functions for the management of contingent scheduling of trains. These tools do not have any decisional capability, so decisions about creation and dispatching of new trains is completely left to human operators. KICKER embeds the functions of the F-TTM tools into a unique system. This is obtained emulating the F-TTM tools, adding decisional knowledge to some of them and realizing new system components that coordinate the needed decisional processes. The KICKER project is thus a contribution to the applied research on intelligent systems that support decision making in the field of inter-modal freight transportation.

KICKER extends the prototype depicted in Section 8.4 by taking into account transportation of *dangerous goods* and *management of yards* along railway lines for taking decisions about the *allocation of the necessary resources*, the *granting of all the needed authorizations* and the *dispatching management* of train circulation. The operative scenario on which KICKER is supposed to operate is the railway line between Bern (Switzerland) and La Spezia (Italy) represented in Figure 21. This line is particularly interesting because it has been identified by European railways as a freightway, heavily interested by inter-modal traffic flows. Such flows originate inside the various container terminals operating along the corridor and terminate in the harbor yards of La Spezia, one of the main Mediterranean doors for container movement.

The results of the KICKER project were satisfactory either for us and for our partners. The proposed approach had a methodological importance for the Italian Railways, due to the lack of application of formal methods for railway systems specification and development. In fact, KICKER represented an example of how a rigorous approach in application design and realization can produce positive results, mainly concerning the exact and quick validation of

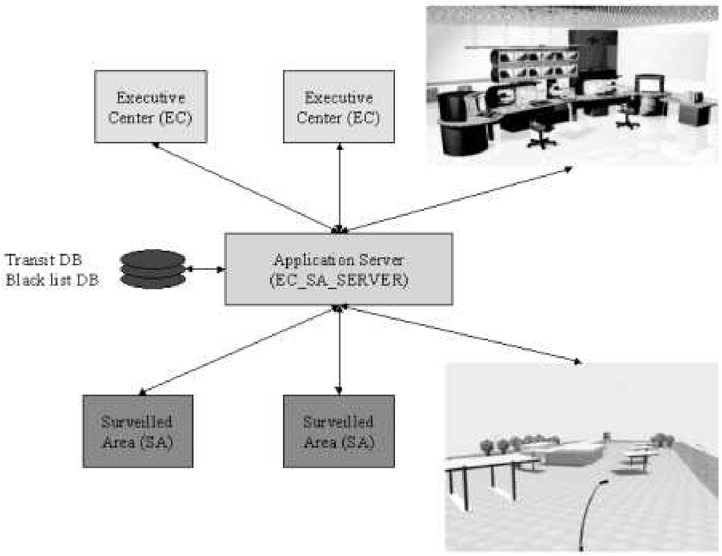


Figure 22. Motorway monitoring application architecture.

system specifications, together with the availability of a simulation platform where it is possible to early testing the system features and performance.

8.6. VEHICLE MONITORING

The most recent application of CaseLP has been the design and development of a working prototype of a vehicle monitoring application in collaboration with Elsag s.p.a. [2].

The prototyped application involves the monitoring of vehicles in the motorway section between Salerno and Reggio Calabria, Italy. The control takes place by means of a distributed network of car detection and license plate reading systems, placed at motorway service areas, sending their transit information to control centers of Police and Financial Police and detecting possible matches with suspect plate numbers. A *transit* is a recognized car plate number with related information, a *black list* is a list of signalled license plate numbers and *alarm* is a transit whose license plate matches an item in a black list. The application includes the physical and software entities represented in Figure 22: *Executive Center* (EC), namely client site of motorway management or Police which allows to browse transits, get alarms, visualize vehicle images, and so on; *Application Server*, which coordinates any communication with the Executive Centers and the Surveillance Areas (see below), dispatches alarms and manages the central transit/alarm database; *Surveillance Area* (SA), which hosts the peripheral intelligent sensors on one or more PCs in a LAN, located at a service station, gets video frames from the local cameras, generates transits and alarms by storing a local black list, and sends transits and alarms to the Application Server.

CaseLP was used to model and test data allocation strategies (for instance, whether to centralize the database of transits or keep it distributed on SAs) and communication strategies. From the prototype execution we got information that we used to compare the distributed and centralized solutions and to understand under which conditions it was preferable one solution or the other one. The most important result we obtained from our simulation runs was that, under reasonable assumptions, the differences between the centralized and distributed solutions are negligible. These results, obtained in less than two person-months, helped in deciding to adopt the centralized solution for the real apparatus. In fact the bandwidth required in this case respected the constraint to be always below the threshold due to satellite communication, and Elsag s.p.a. could reuse already developed components.

9. Related work

CaseLP provides both a MAS development methodology and a framework composed by tools and languages for MAS prototyping. In this section we analyze related work in both fields by surveying and discussing a set of methodologies for AOSE (Section 9.1) and by describing and comparing a set of MAS development frameworks with the CaseLP environment (Section 9.2). Two of the analysed methodologies, MaSE and Prometheus, have a working environment associated with them that can be used to specify and implement the MAS analysed and designed following the methodology. On the other hand, most of the MAS development frameworks described in Section 9.2 have a methodology associated with them to help the MAS developer in reaching her final goal. Since it was not obvious if these integrated environments should be classified as methodologies or development tools, we classified them according to their predominant use. In some cases, such as for the DESIRE framework where the methodological component and the development toolkit component are equally relevant, our classification may be questionable, but we think this does not change the contribution of this related work section.

9.1. Methodologies for AOSE

A lot of work has been done in the past few years to develop clear and well founded methodologies for agent-oriented analysis and design. In this paper we describe six of them, Gaia, ROADMAP, MaSE, Prometheus, MAS CommonKADS and Tropos, and we shortly compare them with the

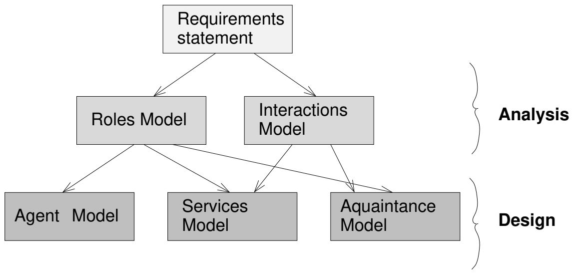


Figure 23. Main models used in Gaia.

CaseLP methodology. The reader can find more information on agent-oriented methodologies in [45].

Gaia [93] is a methodology for agent-oriented analysis and design. It is both general, in that it is applicable to a wide range of multi-agent systems, and comprehensive, in that it deals with both the macro-level (societal) and the micro-level (agent) aspects of systems. Gaia is founded on the view of a multi-agent system as a computational organization consisting of various interacting roles. A Gaian role, whose definition we already quoted in Section 3.1, is characterized by four attributes: responsibilities, permissions, activities, and protocols. The Gaia methodology is appropriate for analysing and designing large-scale real-world applications where: 1) agents are coarse-grained computational systems and are heterogeneous, in that different agents may be implemented using different programming languages, architectures, and techniques; 2) it is assumed that the goal is to obtain a system that maximises some global quality measure, but which may be sub-optimal from the point of view of the system components; 3) the organisation structure of the system is static, as well as the abilities of agents and the services they provide; 4) the overall system contains a comparatively small number of different agent types (less than 100). Gaia is intended to allow an analyst to go systematically from a statement of requirements to a design that is sufficiently detailed that it can be directly implemented. In applying Gaia, the analyst moves from abstract to increasingly concrete concepts. Each successive move introduces greater implementation constraints, and shrinks the space of possible systems that could be implemented to satisfy the original requirements statement. Analysis and design can be thought of as a process of developing increasingly detailed models of the system to be constructed. The main models used in Gaia are summarised in Figure 23. The main Gaian concepts can be divided into two categories: abstract and concrete. Abstract entities are those used during analysis to conceptualise the system, but which do not necessarily have any direct realisation within the system. Concrete entities, in contrast, are used within the design process, and will typically have

Table XI. Abstract and concrete concepts in Gaia.

Abstract concepts	Concrete concepts
Roles	Agent Types
Permissions	Services
Responsibilities	Acquaintances
Protocols	
Activities	
Liveness properties	
Safety properties	

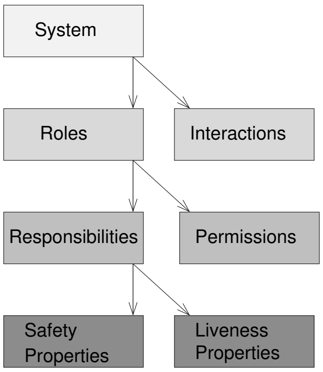


Figure 24. Relationships among the abstract concepts in Gaia.

direct counterparts in the run-time system. Table XI shows these concepts and Figure 24 shows the relationships among the abstract concepts.

An extension of Gaia is the ROADMAP methodology [49] which extends Gaia with the models depicted in Figure 25. The analysis phase in the

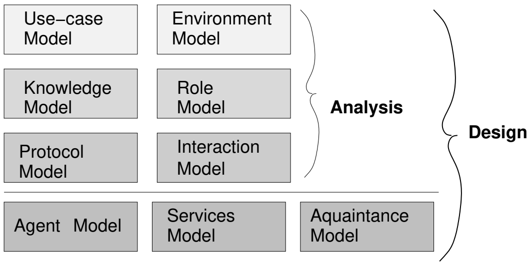


Figure 25. ROADMAP extensions to Gaia models.

original Gaia methodology is extended to cover specifications as well. A use-case model is introduced to support requirements gathering. An environment model and a knowledge model are derived from the use-cases and provide holistic views on the execution environment and the domain knowledge. The collection of protocol descriptions is renamed to the protocol model. An interaction model, based on AUML interaction diagrams [75] is added to model the dynamic aspects of the system. The original role model is extended with a role hierarchy represented as a tree and allowing arbitrary levels of abstraction. The leaf nodes of the tree are atomic roles that represent characteristics of individual agents. All other nodes in the tree are composite roles defined in terms of other roles. A composite role represents a localized organization of agents. Its attributes model social aspects of that organization, such as the organizational structure, social goals, social tasks and social laws. All the analysis models are carried into the design phase. They are optimized towards chosen quality goals and updated to reflect design decisions. The extended role model and the protocol model, which together define the social aspects within the system as well as characteristics of individual agents, are further carried into implementation. Components of these two models, such as roles, are no longer considered abstract. They are concrete first class entities in design and will have a realization in the final implemented system. Similar to roles in Gaia, ROADMAP roles have permissions to access or modify information resources and also the definition of other roles. In summary, following the ROADMAP methodology the system is defined as a computational organization of interacting roles at the analysis stage. The organization is then optimized for quality goals, and populated with agents at the design stage.

The MaSE methodology [29, 27, 26] aims at helping a designer to take an initial set of requirements and analyze, design, and implement a working MAS. This methodology is the foundation for the agentTool development system [28], which also serves as a validation platform. The MaSE methodology, as well as agentTool, is independent of any particular agent architecture, programming language, or communication framework. It specializes more traditional software engineering methodologies by following the phases and steps shown on the right side of Figure 26. The MaSE analysis phase consists of three steps: 1) capturing goals, 2) applying use cases, and 3) refining roles. The design phase has four steps: 1) creating agent classes, 2) constructing conversations, 3) assembling agent classes, and 4) system design. The rounded rectangles denote the MaSE models used to capture the output of each step while the arrows between them show how the models affect each other. Even if it is described as a single flow from top to bottom, in practice the methodology is iterative, allowing the analyst or designer to move between steps and phases freely such that with each successive pass,

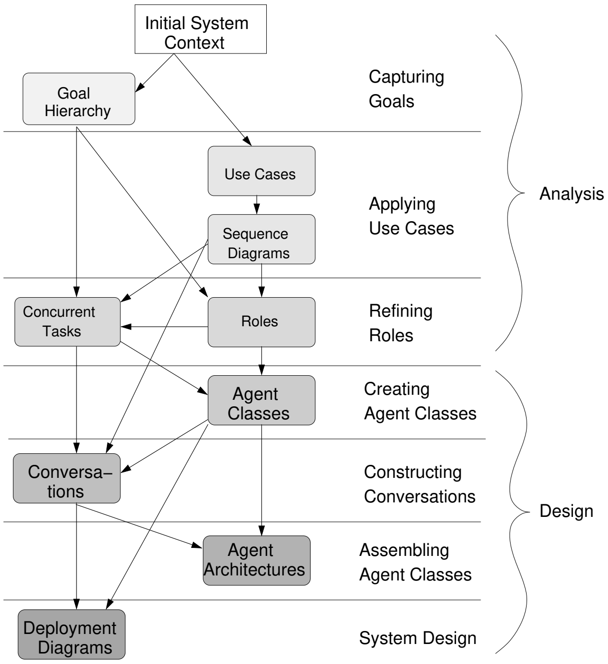


Figure 26. Steps characterizing MaSE methodology.

additional detail is added and, eventually, a complete and consistent system design is produced.

The Prometheus methodology [77] consists of three phases (see Figure 27): 1) system specification, 2) architectural design, and 3) detailed design. System specification consists of identifying functionalities (these are narrow "chunks" of functionality), identifying data read and written by functionalities, prepare functionality schemas, define use cases and derive coupling matrix (both a functionality–functionality interaction matrix, and a functionality–data matrix). Architectural design consists of grouping functionalities to determine agents, defining agent types, assigning percept processing to agents, producing a system level overview diagram, describing the major events, producing agent overview (capabilities, events, actions, data) and designing the capabilities of the agents. Finally, the detailed design involves refining capability internals, introducing plans to handle events and defining details of events, plans and beliefs. Agent Oriented Software Pty. Ltd. has constructed a support tool for the methodology that allows design diagrams to be drawn and generates corresponding skeleton code in Jack (see Section 9.2).

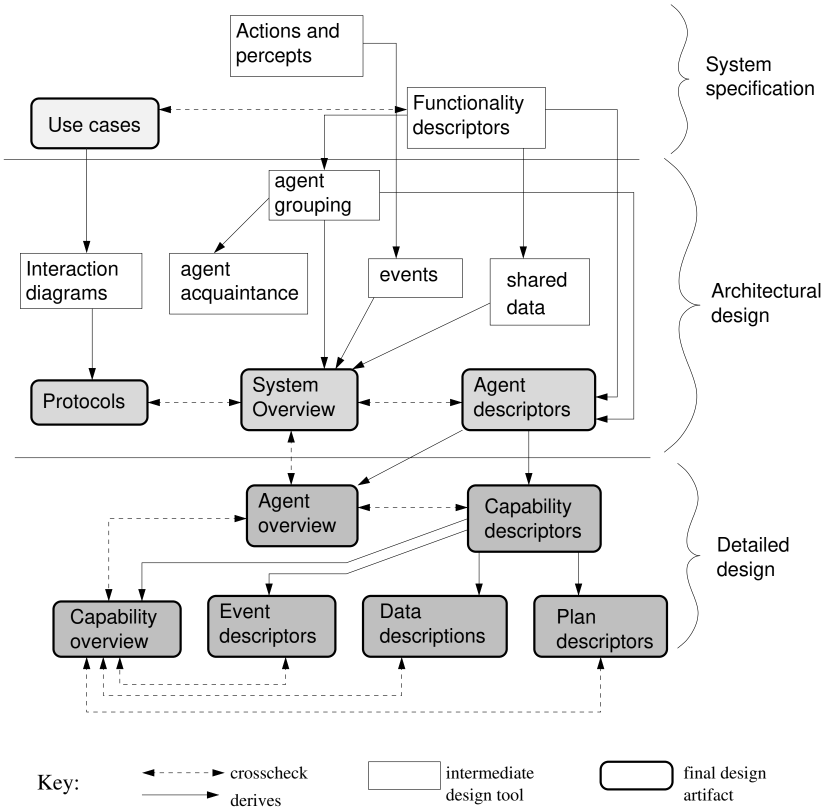


Figure 27. The three phases of Prometheus methodology.

MAS CommonKADS [46] is a method for agent-based software development. It extends CommonKADS, a method for engineering knowledge based systems, with object oriented and protocol engineering techniques. The former ones are used to give structure to the knowledge that agents have, the latter ones permit to specify the interaction and messages exchanged among the agents in the system. The application of the methodology consists of the development of seven *models* that capture differents aspect of the system engineering process:

– *Agent model:* expresses the agent's reasoning capabilities, skills, services.

– *Task model:* describes tasks that agents can perform, their goals, task decomposition, problem solving methods.

– *Expertise model:* describes agent knowledge.

– *Organization model:* describes the organization in which the MAS is going to be introduced.

– *Coordination model:* describes conversations between agents, their interactions, protocols ad required capabilities.

– *Communication model:* details the human-agent interactions.

– *Design 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.

Tropos [79, 43] is based on two key ideas: the notion of agents and all the related mentalistic notions (as beliefs, goals, actions, and plans) are used in all phases of software development; the method covers also the very early phases of requirements analysis and thus allows a deeper understanding of the environment where the software must operate. Tropos supports five phases of software development:

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

– *Late requirements.* The system under realization is described within its operational environment, along with relevant functions and qualities. This description models the system as a number of actors which have a number of 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 ends up with the specification of system's agents.

– *Detailed design.* Each agent in the system is defined 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.

9.1.1. *Discussion on AOSE methodologies*

In [50], T. Juan, L. Sterling and M. Winikoff try to merge the Prometheus and ROADMAP methodologies by isolating a set of general purpose elements which are a common denominator of both methodologies, and by highlightening the differences between them. Since ROADMAP is an extension of Gaia, [50] implicitly includes the merging of the Gaia methodology with the Prometheus one. In this section we try to extend this approach to include CaseLP methodology, MaSE, MAS CommonKADS and Tropos. The result of our attempt is the isolation of a set of core steps that all methodologies face, and a set of general features that the seven methodologies share. Providing a deeper comparison among the methodologies will allow to identify which steps are most carefully faced by each methodology and possibly to integrate all of them by exploiting their best features. In this section we limit ourselves to a simple discussion of the similarities shared by the seven methodologies; a deeper comparison is part of our future work.

The three features that characterize all the methodologies analysed so far and CaseLP methodology are the following ones:

1. all of them are essentially top-down approaches, based on the identification of abstract concepts, be they "roles" or "functionalities", which are progressively refined and decomposed;

2. all of them support an iterative approach to MAS development;

3. all of them are architecture independent at least in the early analysis and design phases.

The basic steps that all the methodologies face are depicted in Table XII, where we show the corresponding models in [50]. In these steps the internal functioning of the agents, and thus their internal architecture, does not come into play; all these steps are architecture-independent, and can be faced by any methodology, also those ones, such as Tropos and Prometheus, that have been designed to build BDI-like agents.

– Agents' capabilities have the very general meaning of functionalities, services, and can be refined into more specific concepts according to the considered methodology. In CaseLP, Gaia, ROADMAP and MaSE capabilities are expressed by roles and are specified during the first stages of the specification development; in MAS CommonKADS they are captured by the agent model, where agents' reasoning skills are analysed. In Prometheus these capabilities are called functionalities and

Table XII. Development steps common to all the methodologies.

Basic step	Correspondent model in [50]
Identify agents' capabilities	Agent Model
Identify agents' types/classes	Agent Model
Identify interconnections among agents' classes	Acquaintance Model
Identify interaction protocols	Interaction Model

are specified during the system specification stage, and in Tropos they have a direct counterpart into the actor capabilities specified during the architectural design phase.

– Agents' types or classes are usually determined during the design phase by grouping capabilities together.

– The interconnections among agents' classes are links connecting two classes: the presence of a link means that there is a communication channel between the two classes or some kind of dependency between them. Messages exchanged by agents connected by a link are *not* specified at this point. In some cases, such as in MaSE, this step is not explicitly separated from the following one, but nevertheless it needs to be faced in order to establish which classes of agents may or may not conversate with other classes.

– Finally, the detailed communication protocols or conversations followed by the agents to exchange information and knowledge are specified during the design phase of each methodology.

Besides these similarities shared by the seven methodologies, there are also some main differences:

– When the MAS development task reaches that point where the internal functioning of the agent must be taken into account, the methodologies propose different approaches: Prometheus and Tropos support the development of BDI-like intelligent agents characterized by goals, plans, beliefs and events, while the other methodologies do not commit themselves to any specific architecture. Even if the latter approach ensures more flexibility, the former one is preferable in case the developer knows in advance that she will adopt a BDI-like architecture for all the agents in the MAS.

– Some methodologies, such as Gaia and ROADMAP, are explicitly conceived to develop MASs where there is a common, final goal. CaseLP

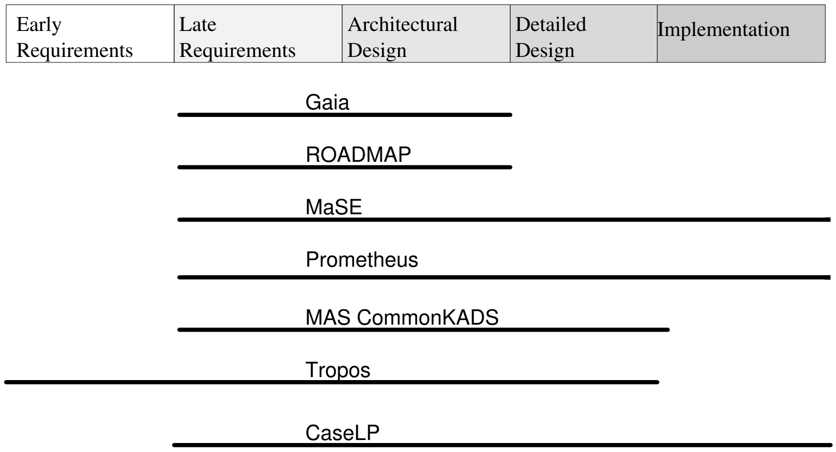


Figure 28. Development process coverage.

methodology has been used to model systems where there is no evident common goal and the agents have opposite interests such as in the fruit-market application, where sellers and buyers both aim at maximizing the amount of money they possess.

- The knowledge analysis is not faced in the same way by all the methodologies. CaseLP has an ontological modeling stage; once modeled, the most suitable ontologies for the application domain are included in the MAS static structure. Gaia does not include the specification of ontologies or of a knowledge model, while its extension ROADMAP does. MAS CommonKADS expertise model specifies the agent knowledge. MaSE and Prometheus do not explicitly face the task of representing knowledge, while Tropos rests on a set of knowledge level concepts [71] that are used uniformly throughout the software development process.

- The development process, involving the five stages of early and late requirements gathering, architectural design, detailed design and implementation, is not covered in the same way by the seven methodologies. In Figure 28, which extends Figure 1 of [43], we show the development process coverage of the methodologies we analysed. The implementation stage refers to the implementation of a working prototype; since ad hoc support tools exist for both MaSE and Prometheus, we say that they support a prototype implementation.

9.2. Frameworks for MAS development

By giving a glance at some sites providing valuable resource about environments for MAS development it appears that there is more than one hundred of agent development environments among which the MAS developer can choose. Given this large amount of toolkits available, it is an hard task to discuss all or most of them; it seems more reasonable to concentrate on a small subset of toolkits which can be analyzed and compared in a deep way. Interesting results have already been achieved following this approach: [34] provides a set of criteria for evaluating development environments, and then applies these criteria to five selected tools and multi-agent systems prototypes: AgentBuilder (`http://www.agentbuilder.com/`), CaseLP, DESIRE [14], IMPACT [37, 36, 4], and Zeus [73]. A very similar approach is taken by [84] which identifies a set of criteria for multi-agent platforms evaluation with a particular focus on methodology, and uses these criteria to evaluate AgentBuilder, Jack [44, 52], MadKit [39] and Zeus. M. Purvis, S. Cranefield, G. Bush and M. Nowostawski [72] compare JADE [6, 5], Zeus and the New Zealand Distributed Information Systems (NZDIS) [82] with respect to the support they provide to communication and ontologies.

In this section we limit ourselves to discuss six environments that are complex and sophisticated enough to represent significant case studies: AgentBuilder, DESIRE, IMPACT, Jack, JADE and Zeus. A short comparison of these toolkits with CaseLP is provided.

AgentBuilder is a commercial product developed by Reticular Systems, Inc. It consists of two major components: the Toolkit and the Run-Time System. The Toolkit includes tools for managing the agent-based software development process, analyzing the domain of agent operations, designing and developing networks of communicating agents, defining behaviors of individual agents, and debugging and testing agent software. The Run-Time System includes an agent engine that provides an environment for execution of agent software. AgentBuilder agents communicate using KQML. All components of both the AgentBuilder Toolkit and the Run-Time System are implemented in Java.

The DESIRE research programme, carried out at the Artificial Intelligence Department of Vrije Universiteit of Amsterdam, Netherland, focuses on the study of compositional multi-agent systems for complex and distributed tasks and development methods for these systems. A major aim of the project is to provide a modeling framework consisting of:

– a design method for compositional multi-agent systems;

- a formal specification language for system design, supporting conceptual design through to detailed (but implementation-independent) design;
- software tools to support system design;
- an implementation generator to automatically translate specifications into code in a specific implementation environment;
- verification tools for static properties of components such as consistency, correctness, completeness.

A number of reusable generic models for agents and tasks have been developed and temporal semantics of reasoning and acting processes has been studied as part of the project.

IMPACT is an international research project led by the University of Maryland. The principal goal of the project is to develop both a theory as well as a software implementation that facilitates the creation, deployment, interaction, and collaborative aspects of software agents in heterogeneous and distributed environments. IMPACT provides a set of servers (yellow pages, thesaurus, registration, type and interface) that facilitate agent interoperability in an application independent manner. It also provides an Agent Development Environment for creating, testing, and deploying agents. The system is developed in Java.

JACK Intelligent Agents by Agent Oriented Software Pty. Ltd. is an agent oriented development environment which includes all components of the Java development environment and offers specific extensions to implement agent behavior. Following the BDI model, JACK intelligent agents are autonomous software components that have explicit goals to achieve or events to handle (desires). The three components of the toolkit are:

- the *JACK Agent Language*, namely the actual programming language used to describe an agent-oriented software system;
- the *JACK Agent Compiler* which pre-processes JACK agent language source files and converts them into pure Java; and
- the *JACK Agent Kernel* which is the runtime engine for programs and provides a set of classes that give JACK programs their agent-oriented functionalities.

JADE is a software framework developed by Telecom Italia Lab for the development of multi-agent applications in compliance with the FIPA agent platform specification [40]. It offers the following features:

- A FIPA-compliant Agent Platform, including the Agent Management System (AMS), the default Directory Facilitator (DF) and the Agent Communication Channel (ACC).
- A distributed agent platform implemented in Java.
- A Java API to send/receive FIPA-ACL messages [41] to/from other agents.
- A library of FIPA interaction protocols ready to be used.
- Support for agent mobility within a JADE agent platform.
- A library to manage user-defined ontologies and content languages.
- A graphical user interface to manage and monitor several agents and agent platforms from the same agent.

Zeus is an environment developed by British Telecommunications, for specifying and implementing collaborative agents, following a clear methodology and using the software tools provided by the environment. In particular, Zeus provides a library of software components and tools that facilitate the rapid design, development and deployment of agent systems. The three main functional components of the Zeus toolkit are

- the Agent Component Library whose components implement the different aspects of agent functionality;
- the Agent Building Tools which provide an integrated development environment through which the agents are specified and generated; and
- the Visualization Tools, namely the runtime environment that enables applications to be observed and, where necessary, debugged.

9.2.1. *Discussion on MAS development frameworks*

The dimensions along which we analyze and compare AgentBuilder, CaseLP, DESIRE, IMPACT, Jack, JADE and Zeus are agent attitudes, social ability, support provided to software engineering, definition of agents and support to MAS development, technical issues and documentation and support available. The reader may refer to [34, 84, 72] for more detailed comparisons.

Agent attitudes. All the seven toolkits allow the development of agents respecting the "basic" definition by N. R. Jennings, K. Sycara and M. Wooldridge discussed in Section 1. As far as more advanced features are concerned, AgentBuilder and Jack agents are conceptualized as BDI agents; in the other cases, agents are not conceptualized in terms of mental notions, but extensions of the tools are provided to cope with this conceptualization (see [10], [11], [32], [53] for CaseLP, DESIRE, IMPACT and Zeus respectively). JADE allows to integrate agents with reasoning capabilities developed with the expert system shell JESS (`http://herzberg.ca.sandia.gov/jess/`). Zeus and Jack provide the developers with the means for easily defining *planning* agents; research is currently under way to extend IMPACT agents with planning capabilities too [70]. Research is also under way to implement *reflective* agents in DESIRE [16] and IMPACT [32]. Jack allows to include, inside a plan, the choice of another plan, thus providing meta-level reasoning capabilities.

Social ability. All the environments implement communication via *message passing*. In AgentBuilder and CaseLP, agents exchange KQML messages. The ACL supported by DESIRE is based on first order logic [15]. IMPACT ACL is called ASDL (Agent Service Description Language) and is based on XML [4]. Jack agent messages are declared in a similar way to Java classes and end up being Java objects to be sent and received. Finally, agents developed with JADE and Zeus exchange FIPA-ACL messages. DESIRE, JADE and Zeus also provide libraries of already developed *interaction protocols* (see [13, 12] for DESIRE, Section 3.5 of [5] for JADE and Sections COORD-1 and COORD-2 of [20] for Zeus).

Software engineering support. All the environments apart from Jack provide tools or languages for defining and using *ontologies*. The only support to ontologies provided by Jack is an object streaming facility that allows to separate the visible terminology of goal exchange from any internal representation. All the environments provide *user interfaces* for configuring, running and debugging the application. Tools for *verification* of properties are provided by DESIRE [38, 48] and CaseLP [9]. CaseLP and Zeus provide a set of *specification languages* among which it is possible to choose the most suitable ones for the different modeling stages. As far as *scalability* is concerned, CaseLP has been tested on applications with more that 200 agents; IMPACT applications usually involve 25-30 agents and Zeus starts showing problems when the agents are more than 100. According to the online documentation, Jack is designed to handle thousands of agents running on relatively low-end hardware.

Definition of agents and support to MAS development. AgentBuilder agents are defined in a language called RADL which extends AGENT-0 [88] and PLACA [90] with object-oriented features. CaseLP agents are defined in ProlAg (Section 6.7). DESIRE provides an unnamed specification language for system design. IMPACT agents are defined using a rule-based language with operators for obligations, permissions, prohibitions, and for accessing external packages [37]. The Jack Agent Language is a super-set of Java extended with constructs to represent agent-oriented features [52]. In a similar way, JADE agents are developed in Java extended with JADE-specific classes implementing different agent behaviors [5]. Zeus agents are programmed by entering their features in the Agent Editor [20]. IMPACT, JADE and Zeus provide standard *utility agents* like directory facilitators, white and yellow page servers for the support to MAS development. DESIRE, CaseLP, and Zeus provide libraries of general reusable components like reusable agent roles, agent architectures, agent skeletons, etc. (see [13], [10, 3, 25], and [20] respectively).

Technical issues. *Concurrency* and *distribution* are supported by all the environments except for CaseLP, which simulates both of them. Research on *mobility* is carried out by the IMPACT, JADE and Zeus teams with promising results. *Security* issues are addressed by IMPACT [7] and JADE [81] from a theoretical point of view, but they are not integrated in the systems yet. *Error-tolerance* issues in IMPACT are studied in [35]. *Complexity* of algorithms for running agents has been deeply studied by the IMPACT team, demonstrating that the performance of the system is good. Efficiency of Jack agents is discussed at the url `http://www.agent-software.com.au/shared/products/faq.html#efficient`, and it seems good. *Efficiency* of CaseLP is quite low, but it should be remembered that the purpose of CaseLP is to develop prototypes and not final applications, thus efficiency is not a central issue. As far as the computer environments of the toolkits are concerned, AgentBuilder, IMPACT, Jack, JADE and Zeus are developed in Java; CaseLP is implemented in both SICStus Prolog and ECLiPSE and DESIRE is implemented Prolog, xpce, C and Java. All the environments have been tested on Windows, and most of them have been tested on Linux, Solaris and IRIX. Jack was also run on a handheld Psion 5mx. The external packages which can be accessed from the toolkits implemented in Java are those for which Java already offers an interface, in particular those ones meeting the JNI interface. DESIRE can access programs that communicate through sockets, files, pipes. CaseLP may access the packages for which ECLiPSE and SICStus Prolog offer interfaces. IMPACT has been used to access different heterogeneous databases. JADE easily integrates JESS agents.

Documentation and support. The best documentation available is probably the one of IMPACT, Jack, and Zeus, containing both user guides and scientific documents. On-line e-mail support is provided to AgentBuilder, Jack, JADE and Zeus users. Some support is also provided by the DESIRE team, while CaseLP and IMPACT provide no on-line support.

10. Conclusions and future work

In this paper we have discussed CaseLP, a rapid prototyping environment which provides a set of tools and languages for developing a working prototype of a MAS following a clear methodology. CaseLP is an instance of the ARPEGGIO open approach for building MAS prototypes where agents may be specified in different languages and external packages may be easily integrated. The main purpose of CaseLP is supporting the heterogeneity of agents specifications and their ability to co-exist within the same running prototype, rather than allowing the integration of external software. These features have been illustrated by means of the example discussed in Section 7 where we used HEMASL for clarifying the BDI abstraction level hierarchy, $\mathcal{E}_{hhf}$ for specifying a BDI concrete class in a language suitable to be translated into ProlAg code and AgentRules for specifying agents with a reactive–proactive architecture. The specification of the marketplace ontology has been given with a textual language, CaseLP-Ontology, and with a graphical language, UML; an extension of UML has also been used to specify the contractProposal protocol. The system architecture has been specified either with UML diagrams and with CaseLP-MASadl. Finally, we have implemented the running prototype by translating the AgentRules and $\mathcal{E}_{hhf}$ specifications into ProlAg code and executing this code by means of the CaseLP Visualizer GUI. The methodology we followed to develop this example supports the developer from the late requirements gathering stage to the implementation of a working prototype by showing how to translate agent specifications written in a high-level language ($\mathcal{E}_{hhf}$ or AgentRules) into executable code. Our methodology tries to fill the gap between the detailed design stage and the implementation stage by blurring the boundaries between executable and non-executable specifications: $\mathcal{E}_{hhf}$ specifications can be executed using the interpreter for the language and then can be refined and compiled into ProlAg code; AgentRules specifications can be easily compiled into the target implementation language. CaseLP methodology supports the fast prototyping of applications which are too complex to be engineered following more traditional approaches and also allows to adopt formal methods during the prototype development to validate those properties that are crucial for the correct functioning of the final application.

To furtherly improve CaseLP we are concentrating our research on the following issues:

- *Integration of new languages for implementing agents and integration of external packages.* We aim at extending the set of implementation languages into which agent specifications can be compiled to allow agents implemented in SICStus Prolog and ECLiPSE to interact with agents written, for example, in Java or C++. We are also working on extending the set of external packages that can be accessed by CaseLP agents.

– *Full integration of UML in CaseLP.* We are deepening our knowledge on UML extensions for modeling roles and interaction protocols and we are working on the development of an automatic compiler for building "shells" of agent programs which respect these protocols and which must be completed by the MAS developer with application-dependent knowledge.

– *Physical distribution and real concurrency of agents.* Finally, we are working at the physical distribution of the agent instances and at their real concurrent execution.

The D-CaseLP (Distributed CaseLP) project started at the beginning of year 2001 to carry out these three tasks. In one year of work a prototypical version of the distributed CaseLP environment has been carefully designed [1] and developed [67] using JADE, and compilers for translating UML diagrams exported as XMI specifications (`http://www/omg.org/technology/xml/`) into JESS pieces of code have been implemented [67]. We have tested the prototypical D-CaseLP environment on a simple example obtaining interesting and promising results.

Acknowledgements

We thank P. E. Maisano, from the Jack support team, for his clear and quick answers to our questions.

References

1. Albertoni, R.: 2001, 'D-CaseLP: un Ambiente Distribuito per l'Integrazione di Agenti Eterogenei'. Master's thesis, Computer Science Department of Genova University, Italy. In Italian.
2. Appiani, E., M. Martelli, and V. Mascardi: 2000, 'A Multi-Agent Approach to Vehicle Monitoring in Motorway'. Technical report, Computer Science Department of Genova University. DISI TR-00-13, Poster session of the 2nd European Workshop on Advanced Video-Based Surveillance Systems, AVBS 2001.
3. Aretti, A.: 1999, 'Semantica di Sistemi Multi-Agente in Logica Lineare'. Master's thesis, Computer Science Department of Genova University, Italy. In Italian.
4. Arisha, K., T. Eiter, S. Kraus, F. Ozcan, R. Ross, and V. Subrahmanian: 1999, 'IMPACT: A Platform for Collaborating Agents'. *IEEE Intelligent Systems* **14**(2), 64–72.
5. Bellifemine, F., G. Caire, and T. Trucco: 2002, 'JADE Programmer's Guide'. Version of February, the 4th, 2002. Downloadable from `http://agent.cs.bath.ac.uk/jade/`.
6. Bellifemine, F., A. Poggi, and G. Rimassa: 2001, 'Developing Multi-agent Systems with JADE'. In: *Intelligent Agents VII*. pp. 89–103. LNAI 1986.
7. Bonatti, P., S. Kraus, and V. S. Subrahmanian.: 1999, 'Secure Agents'. Computer Science Department Technical Report, CS-TR-4068, University of Maryland.

8. Bozzano, M.: 2002, ‘A Logic-Based Approach to Model Checking of Parameterized and Infinite-State Systems’. Ph.D. thesis, Computer Science Department of Genova University, Italy. Technical Report DISI-TH-2002-01.
9. Bozzano, M., G. Delzanno, M. Martelli, V. Mascardi, and F. Zini: 1999a, ‘Logic Programming & Multi-Agent Systems: A Synergic Combination for Applications and Semantics’. In: K. Apt, V. Marek, M. Truszczynski, and D. Warren (eds.): *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag, pp. 5–32.
10. Bozzano, M., G. Delzanno, M. Martelli, V. Mascardi, and F. Zini: 1999b, ‘Multi-Agent Systems Development as a Software Engineering Enterprise’. In: G. Gupta (ed.): *Proc. of First International Workshop on Practical Aspects of Declarative Languages (PADL’99)*. San Antonio, Texas, pp. 46–60. LNCS 1551.
11. Brazier, F., B. Dunin-Keplicz, J. Treur, and L. Verbrugge: 1999a, ‘Modelling Internal Dynamic Behaviour of BDI agents’. In: A. Cesto and P. Schobbes (eds.): *Proceedings of the Third International Workshop on Formal Models of Agents, MODELAGE’97*. Springer Verlag.
12. Brazier, F. M. T., F. Cornelissen, R. Gustavsson, C. M. Jonker, O. L. B. Polak, and J. Treur: 2000a, ‘A Multi-Agent System Performing One-to-Many Negotiation for Load Balancing of Electricity Use’. *International Journal of Electronic Commerce*. To appear.
13. Brazier, F. M. T., F. Cornelissen, C. M. Jonker, and J. Treur: 1999b, ‘Compositional Specification of a Reusable Co-operative Agent Model’. *International Journal of Cooperative Information Systems*. To appear.
14. Brazier, F. M. T., B. M. Dunin-Keplicz, N. R. Jennings, and J. Treur: 1997, ‘DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Frameworks’. *Journal of Cooperative Information Systems* **1**(6), 67–94.
15. Brazier, F. M. T., C. M. Jonker, and J. Treur: 2000b, ‘Compositional Design and Reuse of a Generic Agent Model’. *Applied Artificial Intelligence Journal* pp. 491–538.
16. Brazier, F. M. T. and J. Treur: 1999, ‘Compositional Modelling of Reflective Agents’. *International Journal of Human-Computer Studies*.
17. Chandrasekaran, B., J. R. Josephson, and V. R. Benjamins: 1999, ‘What Are Ontologies, and Why Do We Need Them’. *IEEE Intelligent Systems* **14**(1), 20–26.
18. Ciancarini, P. and M. Wooldridge: 2000, ‘Agent-Oriented Software Engineering: The State of the Art’. In: P. Ciancarini and M. Wooldridge (eds.): *Agent-Oriented Software Engineering - First International Workshop, AOSE 2000*. Limerick, Ireland, pp. 1–28. LNCS 1957.
19. Collins, J. and D. Ndumu: 1999a, ‘ZEUS Methodology Documentation, Part I: The Role Modelling Guide’. Downloadable from `http://more.btexact.com/projects/agents/zeus/`.
20. Collins, J. and D. Ndumu: 1999b, ‘ZEUS Methodology Documentation, Part III: The Application Realisation Guide’. Downloadable from `http://more.btexact.com/projects/agents/zeus/`.
21. Crocé, S.: 1998, ‘Un approccio ad agenti per la simulazione di un modello postale’. Master’s thesis, Computer Science Department of Genova University, Italy. In Italian.
22. Cuppari, A., P. L. Guida, M. Martelli, V. Mascardi, and F. Zini: 1999a, ‘An Agent-Based Prototype for Freight Trains Traffic Management’. In: P. G. Larsen (ed.): *Proc. of FMERail Workshop 5*. Toulouse, France. A satellite workshop of FM’99.
23. Cuppari, A., P. L. Guida, M. Martelli, V. Mascardi, and F. Zini: 1999b, ‘Prototyping Freight Trains Traffic Management Using Multi-Agent Systems’. In: *Proc. of IEEE International Conference on Information, Intelligence and Systems*. Washington, DC.
24. Dart, P., E. Kazmierckaz, M. Martelli, V. Mascardi, L. Sterling, V. Subrahmanian, and F. Zini: 1999, ‘Combining Logical Agents with Rapid Prototyping for Engineering Dis-

tributed Applications'. In: *Proc. 9th International Conference of Software Technology and Engineering (STEP'99)*. Pittsburgh, PA, USA.

25. Degl'Innocenti, R.: 2000, 'UML come Linguaggio per Specificare Agenti: Analisi, Estensione ed Integrazione in CaseLP'. Master's thesis, Computer Science Department of Genova University, Italy. In Italian.
26. DeLoach, S. A.: 2001a, 'Analysis and Design using MaSE and agentTool'. In: *Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001)*. Miami University, Oxford, Ohio.
27. DeLoach, S. A.: 2001b, 'Specifying Agent Behavior as Concurrent Tasks: Defining the Behavior of Social Agents'. In: *5th International Conference on Autonomous Agents*. Montreal, CA.
28. DeLoach, S. A. and M. Wood: 2000, 'Developing Multiagent Systems with agentTool'. In: C. Castelfranchi and Y. Lespérance (eds.): *Intelligent Agents VII*. Boston, MA, USA.
29. DeLoach, S. A., M. F. Wood, and C. H. Sparkman: 2001, 'Multiagent Systems Engineering'. *The International Journal of Software Engineering and Knowledge Engineering* **11**(3).
30. Delzanno, G.: 1997, 'Logic & Object-Oriented Programming in Linear Logic'. Ph.D. thesis, Università di Pisa, Dipartimento di Informatica. Technical Report TD 2/97.
31. Delzanno, G. and M. Martelli: 2001, 'Proofs as Computations in Linear Logic'. *Theoretical Computer Science* **258**(1–2), 269–297.
32. Dix, J., V. Subrahmanian, and G. Pick: 2000, 'Meta Agent Programs'. *Journal of Logic Programming* **46**(1–2), 1–60.
33. eclipse, 'ECLiPSE'. The ECLiPSe Constraint Logic Programming System `http://www.icparc.ic.ac.uk/eclipse/`.
34. Eiter, T. and V. Mascardi: 2001, 'Comparing Environments for Developing Software Agents'. *AI Communications*. To appear.
35. Eiter, T., V. Mascardi, and V. S. Subrahmanian: 2001, 'Error-Tolerant Agents'. In: *Computational Logic: From Logic Programming into the Future*. To appear.
36. Eiter, T. and V. Subrahmanian: 1999, 'Heterogeneous Active Agents, II: Algorithms and Complexity'. *Artificial Intelligence* **108**(1-2), 257–307.
37. Eiter, T., V. Subrahmanian, and G. Pick: 1999, 'Heterogeneous Active Agents, I: Semantics'. *Artificial Intelligence* **108**(1-2), 179–255.
38. Engelfriet, J., C. M. Jonker, and J. Treur: 1999, 'Compositional Verification of Multi-Agent Systems in Temporal Multi-Epistemic Logic'. In: J. P. Müller, M. P. Singh, and A. S. Rao (eds.): *Intelligent Agents V*. Springer-Verlag, pp. 177–194. LNCS 1555.
39. Ferber, J. and O. Gutknecht: 2002, 'MadKit User's Guide – Version 3.1'. `http://www.madkit.org/madkit/doc/userguide/userguide.html`.
40. Foundation for Intelligent Physical Agents: 2001a, 'FIPA Abstract Architecture Specification'. Experimental, 15-08-2001. Downloadable from `http://www.fipa.org/specs/fipa00001/`.
41. Foundation for Intelligent Physical Agents: 2001b, 'FIPA ACL Message Structure Specification'. Experimental, 15-08-2001. Downloadable from `http://www.fipa.org/specs/fipa00061/`.
42. Foundation for Intelligent Physical Agents: 2001c, 'FIPA Agent Software Integration Specification'. Experimental, 15-08-2001. Downloadable from `http://www.fipa.org/specs/fipa00079/`.
43. Giunchiglia, F., J. Mylopoulos, and A. Perini: 2002, 'The Tropos Software Development Methodology: Processes, Models and Diagrams'. In: *Proceedings of the the Third International Workshop on Agent-Oriented Software Engineering, at AAMAS'02*. Bologna, Italy.

44. Howden, N., R. Rönnquist, A. Hodgson, and A. Lucas: 2001, 'JACK Intelligent Agents - Summary of an Agent Infrastructure'. In: *5th International Conference on Autonomous Agents*. Montreal, CA.
45. Iglesias, C. A., M. Garijo, and J. C. González: 1998, 'A Survey of Agent-Oriented Methodologies'. In: J. P. Müller, M. P. Singh, and A. S. Rao (eds.): *Intelligent Agents V*. Paris, France.
46. Iglesias, C. A., M. Garijo, J. C. González, and J. R. Velasco: 1997, 'Analysis and Design of Multiagent Systems using MAS-CommonKADS'. In: M. P. Singh, A. Rao, and M. Wooldridge (eds.): *Intelligent Agents IV*. Providence, Rhode Island, USA. LNAI 1365.
47. Jennings, N. R., K. Sycara, and M. Wooldridge: 1998, 'A Roadmap of Agent Research and Development'. *Autonomous Agents and Multi-Agent Systems* **1**, 7–38.
48. Jonker, C. M., J. Treur, and W. de Vries: 2000, 'Reuse and Abstraction in Verification: Agents Acting in a Dynamic Environment'. In: P. Ciancarini and M. Wooldridge (eds.): *Agent-Oriented Software Engineering - First International Workshop, AOSE 2000*. Limerick, Ireland, pp. 253–268. LNCS 1957.
49. Juan, T., A. Pearce, and L. Sterling: 2002, 'ROADMAP: Extending the Gaia Methodology for Complex Open Systems'. In: *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*. Bologna, Italy, pp. 3–10.
50. Juan, T. and L. S. M. Winikoff: 2002, 'Assembling Agent-Oriented Software Engineering Methodologies from Features'. In: *Proceedings of the the Third International Workshop on Agent-Oriented Software Engineering, at AAMAS'02*. Bologna, Italy.
51. Kendall, E. A.: 1998, 'Agent Roles and Role Models: New Abstractions for Intelligent Agent System Analysis and Design'. In: *International Workshop on Intelligent Agents in Information and Process Management*. Germany.
52. Maisano, P. E.: 2002, 'JACK Intelligent Agents^{TM} User Guide – Release 3.5'. Downloadable from `http://www.agent-software.com.au/shared/demosNdocs/jack/html/index.html%`.
53. Mangina, E. et al: 2001, 'MUlti Agent SYstem Knowledge Representation For Power Plant'. In: *Proc. of the International Conference on Intelligent System Application to Power Systems*. Budapest, Hungary.
54. Marini, S.: 1999, 'Specifica di Sistemi Multi-Agente Eterogenei'. Master's thesis, DISI - Università di Genova, Italy. In Italian.
55. Marini, S., M. Martelli, V. Mascardi, and F. Zini: 2000a, 'HEMASL: A Flexible Language to Specify Heterogeneous Agents'. In: A. Corradi, A. Omicini, and A. Poggi (eds.): *Proceedings of WOA 2000. Dagli Oggetti Agli Agenti*. Parma, Italy.
56. Marini, S., M. Martelli, V. Mascardi, and F. Zini: 2000b, 'Specification of Heterogeneous Agent Architectures'. In: C. Castelfranchi and Y. Lespérance (eds.): *Intelligent Agents VII*. Boston, MA, USA.
57. Martelli, M., V. Mascardi, and F. Zini: 1997a, 'Applying Logic Programming to the Specification of Complex Applications'. *Mathematical Modelling and Scientific Computing* **8**.
58. Martelli, M., V. Mascardi, and F. Zini: 1997b, 'Applying Logic Programming to the Specification of Complex Applications'. In: M. Falaschi, M. Navarro, and A. Policriti (eds.): *Proc. of Appia–Gulp–Prode 1997: Joint Conference on Declarative Programming*. Grado, Italy, pp. 491–499. Extended version.
59. Martelli, M., V. Mascardi, and F. Zini: 1997c, 'CaseLP: A Complex Application Specification Environment Based on Logic Programming'. In: *Proc. of ICLP'97 Post Conference Workshop on Logic Programming and Multi-Agents*. Leuven, Belgium, pp. 35–50.
60. Martelli, M., V. Mascardi, and F. Zini: 1998, 'Towards Multi-Agent Software Prototyping'. In: H. S. Nwana and D. T. Ndumu (eds.): *Proc. of The 3rd International Conference*

and Exhibition on The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'98). London, UK, pp. 331–354.

61. Martelli, M., V. Mascardi, and F. Zini: 1999a, 'A Logic Programming Framework for Component-Based Software Prototyping'. In: A. Brogi and P. Hill (eds.): *Proc. of 2nd International Workshop on Component-based Software Development in Computational Logic (COCL'99)*. Paris, France.
62. Martelli, M., V. Mascardi, and F. Zini: 1999b, 'Specification and Simulation of Multi-Agent Systems in CaseLP'. In: M. C. Meo and M. Vilares-Ferro (eds.): *Proc. of Appia–Gulp–Prode 1999: Joint Conference on Declarative Programming*. L'Aquila, Italy, pp. 13–28.
63. Mascardi, V.: 1996, 'Un Sistema Multi-Agente Logico'. Master's thesis, Computer Science Department of Genova University, Italy. In Italian.
64. Mascardi, V.: 2002, 'Logic-Based Specification Environments for Multi-Agent Systems'. Ph.D. thesis, Computer Science Department of Genova University, Italy. DISI-TH-2002-04. Downloadable from `http://www.disi.unige.it/person/MascardiV/`.
65. Mascardi, V. and E. Merelli: 1999, 'Agent-Oriented and Constraint Technologies for Distributed Transaction Management'. In: *Proc. of IIA and SOCO*. Genova, Italy.
66. Mayfield, J., Y. Labrou, and T. Finin: 1995, 'Evaluation of KQML as an Agent Communication Language'. In: *Intelligent Agents II*. Springer Verlag. LNAI 1037.
67. Miglia, S.: 2002, 'Specifica ed Implementazione di Ruoli e Protocolli d'Interazione per Agenti in D-CaseLP'. Master's thesis, Computer Science Department of Genova University, Italy. In Italian.
68. Miller, D.: 1996, 'Forum: A Multiple-Conclusion Specification Logic'. *Theoretical Computer Science* **165**(1).
69. Müller, J. P.: 1999, 'The Right Agent (Architecture) to Do the Right Thing'. In: J. P. Müller, M. P. Singh, and A. S. Rao (eds.): *Intelligent Agents V*. Springer-Verlag, pp. 211–225. LNCS 1555.
70. Munoz-Avila, H., J. Dix, D. S. Nau, and Y. Cao: 2000, 'IMPACTing SHOP: Planning in a Multi-Agent Environment'. In: K. Satoh and F. Sadri (eds.): *CL-2000 Workshop on Computational Logic in Multi-Agent Systems 2000 (CLIMA-00)*. London, UK, pp. 44–55.
71. Newell, A.: 1982, 'The Knowledge Level'. *Artificial Intelligence* **18**, 87–127.
72. Nowostawski, M., G. Bush, M. Purvis, and S. Cranefield: 2000, 'Platforms for Agent-Oriented Software Engineering'. In: *Proceedings of the Seventh Asia Pacific Software Engineering Conference (APSEC'2000)*. Los Alamitos, CA: IEEE Computer Society Press, pp. 480–488.
73. Nwana, H., D. Ndumu, L. Lee, and J. Collis: 1999, 'ZEUS: A Tool-Kit for Building Distributed Multi-Agent Systems'. *Applied Artifical Intelligence Journal* **13**(1), 129–186.
74. Odell, J., H. Parunak, and B. Bauer: 1999, 'Response to the OMG Analysis & Design Task Force UML 2.0 Request for Information – Extending UML for the Specification of Agent Interaction Protocols'.
75. Odell, J., H. V. D. Parunak, and B. Bauer: 2000, 'Extending UML for Agents'. In: *Proceedings of the Agent-Oriented Information System Workshop at the 17th National Conference on Artificial Intelligence*.
76. P. L. Guida (Project Coordinator): 1998, 'EuROPE–TRIS Project — 4th European Union RDT Program Telematics Applications Programme'. Transport (Rail) Contract DGXIII - TR 1022 (TR), Specifications and System Design, Deliverable D04.2. Web page: `http://www.srd.it/TRISprj`.

77. Padgham, L. and M. Winikoff: 2002, 'Prometheus: A Methodology for Developing Intelligent Agents'. In: *Proceedings of the the Third International Workshop on Agent-Oriented Software Engineering, at AAMAS'02*. Bologna, Italy.
78. Pedrini, M. D.: 1998, 'CaseLP Visualizer: un Tool di Visualizzazione per Sistemi Multi-Agente Logici'. Master's thesis, Computer Science Department of Genova University, Italy. In Italian.
79. Perini, A., P. Bresciani, F. Giunchiglia, P. Giorgini, and J. Mylopoulos: 2001, 'A Knowledge Level Software Engineering Methodology for Agent Oriented Programming'. In: *5th International Conference on Autonomous Agents*. Montreal, CA.
80. Persano, G.: 1998, 'Gestione Distribuita di Informazioni Mediche Mediante Tecniche Multi-Agente'. Master's thesis, Computer Science Department of Genova University, Italy. In Italian.
81. Poggi, A., G. Rimassa, and M. Tomaiuolo: 2001, 'Multi-User and Security Support for Multi-Agent Systems'. In: *Proceedings of WOA 2001 Workshop. Dagli Oggetti Agli Agenti*. Modena, Italy.
82. Purvis, M., S. Cranefield, G. Bush, D. Carter, B. McKinlay, M. Nowostawski, and R. Ward: 2000, 'The NZDIS Project: an Agent-based Distributed Information Systems Architecture'. In: J. R. H. Sprague (ed.): *CDROM Proceedings of the Hawaii International Conference on System Sciences (HICSS-33)*.
83. Rao, A. S. and M. Georgeff: 1995, 'BDI Agents: from Theory to Practice'. In: *Proc. of International Conference on Multi Agent Systems (ICMAS'95)*. San Francisco, CA, USA.
84. Ricordel, P.-M. and Y. Demazeau: 2000, 'From Analysis to Deployment: a Multi-Agent Platform Survey'. In: *ESAW'00 – First International Workshop on Engineering Societies in the Agents' World*. Berlin, Germany, pp. 93–105. LNAI 1972.
85. Riehle, D.: 1997, 'A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose'. Technical report, Union Bank of Switzerland, Zurich, Switzerland. Ubilab 97.1.1.
86. Robinson, J. A.: 1971, 'Computational Logic: The Unification Computation'. *Machine intelligence* **6**, 63–72.
87. Rumbaugh, J., I. Jacobson, and G. Booch: 1998, *The Unified Modeling Language Reference Manual*. Addison-Wesley.
88. Shoham, Y.: 1993, 'Agent-oriented programming'. *Artificial Intelligence* **60**, 51–92.
89. sicstus, 'SICStus Prolog'. SICS AB. `http://www.sics.se/sicstus.html`.
90. Thomas, S. R.: 1995, 'The PLACA Agent Programming Language'. In: *Intelligent Agents. Agent Theories, Architectures, and Languages - 1th International Workshop*.
91. Vinosky, S.: 1997, 'CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments'. *IEEE Communications Magazine* **14**(2).
92. Wooldridge, M. and N. R. Jennings: 1995, 'Intelligent Agents: Theory and Practice'. *The Knowledge Engineering Review* **10**(2), 115–152.
93. Wooldridge, M., N. R. Jennings, and D. Kinny: 2000, 'The Gaia Methodology for Agent-Oriented Analysis and Design'. *International Journal of Autonomous Agents and Multi-Agent Systems* **3**(3), 285–312.
94. Zini, F.: 2001, 'CaseLP, a Rapid Prototyping Environment for Agent-Based Software'. Ph.D. thesis, Computer Science Department of Genova University, Italy. DISI-TH-2001-03. Downloadable from `http://sra.itc.it/people/zini/`.
95. Zini, F. and L. Sterling: 1999, 'Designing Ontologies for Agents'. In: M. C. Meo and M. Vilares-Ferro (eds.): *Proc. of Appia–Gulp–Prode 1999: Joint Conference on Declarative Programming*. L'Aquila, Italy, pp. 29–42.
96. Ziu, R.: 1997, 'Modello in Programmazione Logica di una Logistica Intermodale'. Master's thesis, Computer Science Department of Genova University, Italy. In Italian.