DCaseLP: a Prototyping Environment for Multi-Language Agent Systems*

Ivana Gungui, Maurizio Martelli and Viviana Mascardi

Dipartimento di Informatica e Scienze dell'Informazione – DISI,
Università di Genova, Via Dodecaneso 35, 16146, Genova, Italy.
iva_sim@yahoo.it, {martelli,mascardi}@disi.unige.it

Abstract. This paper describes DCaseLP, a multi-language development environment for Multi-Agent Systems. DCaseLP provides tools and languages for modelling and implementing a MAS prototype following a set of steps which guide the developer from the late requirement analysis to the prototype implementation. Full support for validating the MAS model by running the prototype in the JADE platform is offered. DCaseLP and its ancestor, CaseLP, have been employed to develop many applications also in collaboration with Italian companies, thus demonstrating the feasibility of the proposed approach.

1 Introduction and Motivation

The correct and efficient engineering of heterogeneous, distributed, open, and dynamic applications is one of the technological challenges faced by Agent-Oriented Software Engineering (AOSE). Researchers and practitioners agree that engineering a software system involves a non negligible amount of risk of different kinds [4]. This is particularly true when the system to engineer is as complex as a MAS. However, the risk intrinsic to the development of a MAS could be mitigated by following a prototyping approach.

In fact, a prototype is not something to be delivered to a client, usually. One of the reasons is precisely the purpose of the prototype: proof of concept. It intends to show the client what the final software product will look like, in order to gain a full understanding of the client's requirements before starting the implementation of the product. Developing a working prototype does not usually require a great deal of time. This means early availability of a product that the customer can evaluate, and the opportunity to detect any possible inaccuracies. Early detection of inadequacies allows to keep the cost of the software prototype much lower than the cost of the end product, and this reduces the financial risks involved in the development process. The iterative prototyping process ensures the flexibility to revise the requirements or critical design choices several times before committing to any final decision. Finally, efficiency is not a key feature: a prototype does not need to be extremely efficient, and therefore it can be produced using methods and tools that are suitable for validation of the initial requirements, but not necessarily for the implementation of the final product.

* This work was partially supported by the research project "Iniziativa Software CINI - Finmeccanica".

In this paper we describe the DCaseLP framework for MAS prototyping. DCaseLP stands for *Distributed Complex Applications Specification Environment based on Logic Programming*. Although initially born as a logic-based framework, as the acronym itself suggests, DCaseLP has evolved into a multi-language prototyping environment that integrates both imperative (object-oriented) and declarative (rule-based and logic-based) languages, as well as graphical ones. The rationale behind DCaseLP is that MAS development requires engineering support for a diverse range of non-functional properties, such as understandability of the MAS at various conceptual levels, integrability of heterogeneous agent architectures, usability, re-usability, and testability. Creating one monolithic AOSE approach to support all these properties is not feasible. Rather, we expect different approaches to be suitable for modelling, verifying, or implementing various properties. By providing the MAS developer with a set of languages, and allowing for the choice of the most suitable one to model, implement, and test each property, DCaseLP heads towards the modular approach to AOSE proposed by [17,18]. These ideas, that we applied since the beginning of our project in 1996,[1] are currently gaining a wide consensus also for the final product development and maintenance stages [16].

The languages and tools that DCaseLP integrates are UML and an XML-based language for the analysis and design stages, Java, JESS and tuProlog for the implementation stage, and JADE for the execution stage. Software libraries for translating UML class diagrams into code and for integrating JESS and tuProlog agents into the JADE platform are also provided.

There are many motivations behind supporting these languages as part of an integrated environment:

1. The ability to describe the MAS's architecture and interaction strategies in UML[2] may be exploited by any average skilled software developer that knows how to draw UML class and sequence diagrams and wants to generate code starting from those diagrams, without needing a deep knowledge of the language in which code is generated. In fact, the generated code contains comments that explain what should be added inside the code in order to make it executable, and make code completion easy to be faced.

2. JESS, the Java Expert System Shell [13], allows the developer to supply knowledge in the form of declarative rules that are processed by means of the Rete algorithm [11]. It is a very expressive and concise language, suitable for implementing lightweight and fast expert agents that may easily access and reason about Java objects.

3. Computational logic and logic programming in particular are very suitable to implement sophisticated, self-aware agents able to reason about themselves and the other agents in the MAS [20]. tuProlog [10] provides a light Prolog engine written in Java. It may be used to build rational agents that behave according to the "strong" agent notion, namely entities conceptualised in terms of mental attitudes and able to perform some reasoning about their mental state.

---

[1] At that time, the project name was CaseLP – without *D*, since no support to distribution was given yet.

[2] At the time of writing, only the translation from UML class diagrams to code is fully supported by DCaseLP. We have already developed a separate translator from sequence diagrams to Prolog agent skeletons, `http://www.disi.unige.it/person/MascardiV/Software/WEST2EAST.html`, and we are currently integrating it in DCaseLP.

4. Finally, as far as the importance and usefulness of JADE is concerned, we may quote [2] that describes JADE as "*probably the most widespread agent-oriented middleware in use today.*"

The paper is organised in the following way: Section 2 discusses the AOSE stages that DCaseLP addresses; Section 3 describes the libraries that DCaseLP provides to the user; Section 4 discusses the most recent applications of DCaseLP; Section 5 compares DCaseLP with relevant related tools; and finally Section 6 concludes.

2 DCaseLP: an Integrated AOSE Approach and Environment

DCaseLP provides the languages and tools that support a MAS developer in the engineering stages from late requirements analysis to prototype testing. In the following sections we outline these engineering stages. The suggested AOSE approach is based upon existing proposals.

2.1 Modelling stage (analysis and design)

The analysis stage is mainly role-driven. We share the belief that roles are the key abstraction in MAS modelling with several researchers in the AOSE field. Role modelling allows the MAS developer to specify *what* the system can do, without going into the details about *how* the system will do it. Roles are played by agent classes. To make an example, *Seller* and *Buyer* are two roles that may be played by the *fruitSeller* and *fruitBuyer* agent classes respectively, as well as by a *fruitExchanger* class that plays both of them. In order to define roles and interactions taking place among them, the MAS developer may follow the guidelines given in [5], where modularity, high cohesion, parsimony, completeness, and low coupling are used as characterising criteria for qualifying roles.

Once the role model is well understood, the developer needs to define how communication among entities playing different roles takes place; which roles should be assigned to which agent class; and how many instances of each agent class are required for the given application.

The language that DCaseLP provides to the user in order to cope with these aspects is UML, along the lines of [3,5]. The tool that allows the integration of role models defined using UML into DCaseLP consists of a set of XSL configuration files that define the rules for translating XMI representations of UML diagrams into executable code. The first issue to address refers to the activity of defining interactions among the roles needed in the MAS under development. A "Protocol Diagram" may be defined to this aim. A suitable notation is provided by UML sequence diagrams where - according to the AUML philosophy [22] - roles are used instead of classes or objects as the entities involved in the interaction.

In order to identify the agent classes needed by the application, and assign roles to them, the developer may consider that both access points for information, expertise, and services, and entities that are responsible for controlling some kind of activity, are good agent class candidates [5]. This assignment of roles to classes may be modelled as an "Architecture Diagram", namely, a UML class diagram where UML classes may be

either agent roles or agent classes (according to their stereotype), and "plays" relationships between agent classes and agent roles are defined. For each agent class defined in the modelling stage, the corresponding code implementing the class behaviour should be defined in the implementation stage.

The number of instances of each agent class depends on the MAS under development. Clearly decoupling agent classes from agent instances enforces the modularity and re-usability of the agent class model. Agent instances are assigned to their corresponding agent classes in the "Agent Diagram", a UML class diagram that includes agent classes and agent instances. When the MAS is going to be implemented, the initial state of each agent instance must be defined by encoding it in the chosen language.

2.2 Implementation Stage

The implementation of the MAS prototype must be coherent with the specification given in the previous step. Ensuring this coherence is a demanding task for the developer of the prototype, but DCaseLP may reduce this burden by providing a semi-automatic translator from UML into JESS, that is one of the implementation languages offered by DCaseLP.

During the implementation stage, the *integration of external software* comes into play. To this aim, DCaseLP follows the *Wrapper Agent* model, [12], defined by the Foundation for Intelligent Physical Agents, FIPA, with agents that act as "wrappers " for the external pieces of code. The external packages which can be accessed by the prototype are all the ones that Java, tuProlog and JESS provide interfaces for.

DCaseLP provides a set of libraries for integrating in JADE agents whose behaviour is entirely programmed in tuProlog or JESS, and not simply a way of executing tuProlog or JESS pieces of code from inside JADE agents. From a developers point of view, the difference is substantial. By "integrating tuProlog and JESS agents into JADE", we mean the ability to specify the complete behaviour of the agent, including its ability to communicate with other agents running into a JADE platform, in tuProlog and JESS and, then, execute these specifications. A developer that is able to write tuProlog or JESS code, but that is not able to program in Java (and thus, is not able to define JADE agents), can define active and communicating agents, and run them in JADE, without even knowing the structure and definition of JADE agents. This is possible because we have extended both tuProlog and JESS with primitives that allow them to communicate with agents running in a JADE platform (no matter if they are tuProlog, JESS, or pure JADE agents) in a completely transparent way. This is obviously different from, and more sophisticated than, providing the means to integrate code into JADE agents, but still constraining the developer to use and know the JADE package.

2.3 Execution Stage

The execution of the MAS prototype allows the MAS developer to test and evaluate the analysis, design, and implementation choices that were made during development.

We identified a set of general evaluation criteria, which are relevant for most MAS applications. The monitoring and debugging tools that JADE offers can be used to evaluate them:
– *Load Balancing and Load Peaks*. The amount of work done by each agent in the prototype can be monitored by measuring the number of exchanged messages (for example, by using the Sniffer agent provided by JADE). By means of these measurements, the developer can identify the overloaded agents and may then decide to modify the architecture of the MAS, for example by defining a different assignment of roles to agent classes.
– *Correctness of communication protocols*. Implementation of the communication protocols can be tested by monitoring which messages are received by which agent, and whether there are agents that receive messages that they do not understand. If this is the case, there may be a mis-implementation of the protocols related to the roles the agents must play in the MAS.
– *MAS Topology*. During the simulation, the "neighbours" of each agent, i.e., the set of agents it can exchange messages with, can be set up. This allows the developer to experiment various interconnection topologies.

3 Using DCaseLP

DCaseLP has been implemented in order to provide

1. a *support to the steps* described in Section 2, and
2. a *transparent integration* between Java, JESS, and tuProlog agents running in a JADE platform (Figure 1).

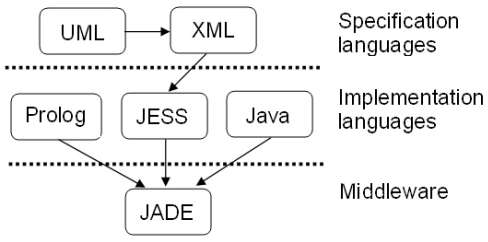

Fig. 1. DCaseLP packages.

The result of our work consists of the following packages:

1. the Java `UMLInJADE` package that contains the Java classes and the XSL style sheets for translating UML diagrams created with any UML modelling tool[3] and exported into XMI, into (ad-hoc) intermediate XML models and, from these, to create the files containing the code for running the JESS agents into JADE.

[3] In our experiments, we used ArgoUML.

2. the Java `jessInJADE` package, that contains the classes that implement JESS *agents*, to be run in the JADE environment, and whose behaviour is fully specified by means of the JESS language.
3. the Java `tuPInJADE` package, that contains the classes that implement tuProlog *agents*, to be run in the JADE environment, and whose behaviour is fully specified by means of the tuProlog language.

The three packages, together with their manuals and tutorials, are available from `http://www.disi.unige.it/person/MascardiV/Software/DCaseLP.html`. Examples of use of DCaseLP are also provided from the above URL, together with the code for the electronic commerce application discussed in Section 4.

3.1 The UMLInJADE package

The `UMLInJADE` package provides the means to translate, in a semi-automatic way, the high level specification of the MAS, consisting of the Protocol, Architecture and Agent diagrams introduced in Section 2, given either in UML (usable only for Architecture and Agent diagrams) or in an XML intermediate format (available for all of them), into JESS agents.

The protocol diagram sets the interaction rules among roles that will be played by classes of agents. There is only one way to specify protocol diagrams in a format that can be automatically translated into code, namely, using our XML intermediate format. Currently, we cannot define protocol diagrams directly in ArgoUML (`http://argouml.tigris.org`), which is the UML editor that we currently use for drawing UML diagrams and for exporting them into XMI, because ArgoUML does not support the definition of UML sequence diagrams (and protocol diagrams are expressed using the same notation of sequence diagrams). Other UML editors such as Poseidon (`http://www.gentleware.com/index.php`), that support the definition of sequence diagrams, export them into an XMI format that is not compliant with our translation program. We are currently working to a definition of a new translation program from XMI to our XML intermediate format, that is compliant with Poseidon, in order to overcome this limitation of the current release of DCaseLP.

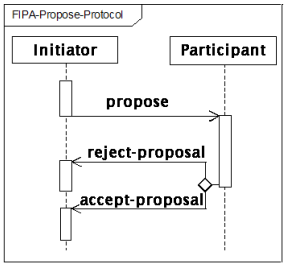

Fig. 2. FIPA propose protocol.

To show how our intermediate XML looks like, we use it to describe the propose protocol suggested by FIPA (Figure 2), where a Seller role substitutes the role of Initiator, and a Buyer role substitutes the role of Participant; a fragment of the resulting intermediate XML specification is shown below. The structure of our XML notation is trivial, with roles characterised by a role name, and the ordered list of messages that they send or receive, eventually embedded into "or", "xor" and "and" tags.

```
1   <protocoldiagram>
2    <role><name>Seller</name>
3     <msgs><msg><sender>Seller</sender>
4      <receiver>Buyer</receiver>
5      <act>PROPOSE</act></msg>
6     <xor-thread>
7      <thread><msg>
8       <sender>Buyer</sender><receiver>Seller</receiver>
9       <act>REJECT_PROPOSAL</act></msg></thread>
10      <thread><msg>
11       <sender>Buyer</sender><receiver>Seller</receiver>
12       <act>ACCEPT_PROPOSAL</act></msg></thread>
13     </xor-thread></msgs></role>
.....
n </protocoldiagram>
```

The architecture diagram expresses the relationships that exist between roles and classes of agents. It can be specified either by means of a UML class diagram like the one shown in Figure 3, or in the XML intermediate format.

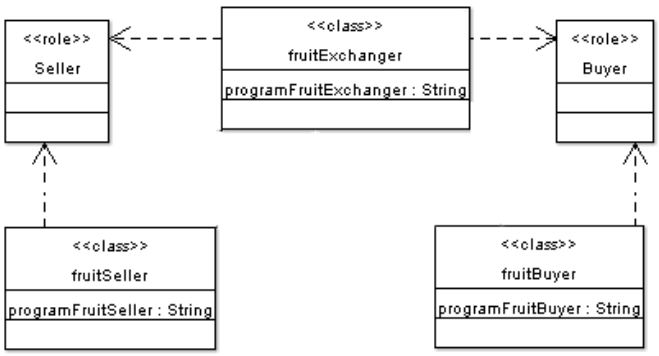

Fig. 3. An architecture diagram.

The agent diagram states which agent classes have which instances. Like the architecture diagram, it can be specified either by means of a UML class diagram (like the one shown in Figure 4), or in intermediate XML format. The `UMLInJADE` package

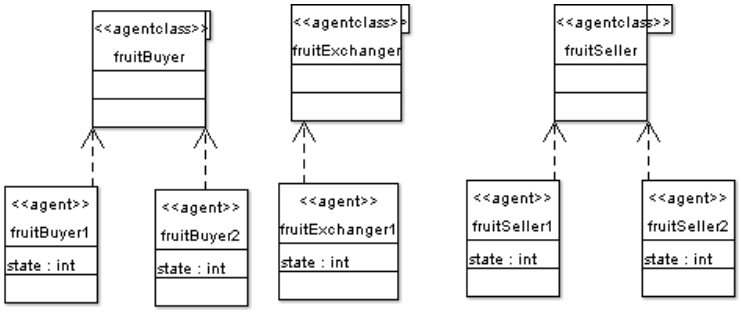

Fig. 4. An agent diagram.

defines the `Specif2Code` class, that is used to perform the translation from the high level description of the MAS (either given by means of UML and exported into XMI, or by means of the intermediate XML format) into partial JESS agents that behave according to the interaction protocols, and are organised according to the architecture and agent diagrams. The Java code necessary to load and run a JESS agent into a JADE platform is also automatically generated.

The usage of `UMLInJADE.Specif2Code` is simple: from a command line, the developer just needs to type in `javaUMLInJADE.Specif2Code` and enter the information on the location of the XMI or XML files to translate, that are input by means of a set of interactive windows.

In order to execute the MAS resulting from the translation of the high level specification, the generated JESS code, that we will name JESS "skeleton" and that is put by the translator into a directory named jecode, must first be completed (see Section 3.2). The completed JESS code must be kept in the directory where it was generated and its name must not be changed, for ensuring its integration into JADE.

Once all the JESS skeletons have been completed, the JADE MAS can be built and its simulation can start. First of all, the Java "stubs" necessary for integrating the JESS skeletons into JADE, and automatically generated by the translation program taking the architecture diagram into account, must be compiled. Supposing that the Java stubs are compiled into the jacode directory, and that the MAS agent diagram is the one specified in Figure 4, then the MAS simulation in JADE can be started by entering the command `java jade.Boot fB1:fruitBuyer fB2:fruitBuyer fS1:fruitSeller fS2:fruitSeller fE1:fruitExchanger` from the jacode directory.[4] This command launches a JADE platform (first argument, `jade.Boot`) containing an agent named `fB1`, and whose behaviour is given by the JESS program integrated in JADE by the Java `fruitBuyer` stub. In the same way, there are an agent instance `fB2` with class `fruitBuyer`, an instance `fS1` with class `fruitSeller`, and so on. The names of the agent classes are directly obtained by the architecture diagram. The agent diagram has a one-to-one correspondence with the command line typed to start the simulation. In fact, the command line has one argument for each

[4] We have substituted `fruitBuyer1`, etc, with `fB1`, etc, for readability.

agent instance specified in the diagram, and the argument consists of the agent name separated by a colon from the agent class.

3.2 The jessInJADE package

The `jessInJADE` package defines two classes, `jessAg` and `jessBhv`. Every JESS agent must be defined by means of a Java class that extends `jessAg`, which, in turn, extends the JADE `Agent` class by adding to it the capability, for an agent written in JESS, to exchange messages with any JADE agent. The class `jessBhv` defines the behaviour of a JESS agent.

In order to integrate a JESS piece of code into a JADE agent, we provide both the skeleton of a JESS agent, and the Java stub that is necessary to integrate the JESS agent into JADE. The `JavaStubSkeleton` code can be found in the `jessInJADE` directory. Once opened in a text editor and completed (self-explaining comments in the code indicate where the developer must add his/her own code), the `JavaStubSkeleton` behaves like a JADE agent whose only activity is to execute the JESS code obtained by editing and completing the `jessAgentSkeleton` file. This file (also found in the `jessInJADE` directory) must be edited by the developer, and completed with the JESS rules and initial facts that characterise the agent's behaviour and initial beliefs.

If the developer takes advantage of the translation process from the UML and XML specifications of protocol and architecture diagrams into JESS code, one Java stub and one JESS skeleton are automatically created for each agent class involved in the MAS. In other words, the JESS rules that characterise the agent's behaviour do not need to be encoded by the developer, since they are automatically generated in order to comply with the protocol given in the XML intermediate format. Since the protocol specifies neither the agent's initial state, nor the conditions under which a message is sent, the developer still needs to manually complete the code, but his/her work is less, and easier, than writing a JESS agent from scratch. On the other hand, the Java stub that is generated by the `Specif2Code` method, is ready to use and does not need to be edited.

The built-in predicates defined by the `jessInJADE` package include a `send` function, that sends an `ACLMessage` to an agent running in a JADE platform, and a `receive` function. The input of the `send` function is the fact `ACLMessage` whose template is predefined in any `JessAg`. The slots defined for the `ACLMessage` fact are the same as the one present in a JADE ACLMessage, namely communicative act (or "performative"), content of the message, sender, receiver, and eventually other arguments. The `receive` function returns a reference to the first `ACLMessage` available in the mail box of the agent, and `nil` if no message is available.

3.3 The tuPInJADE package

The `tuPInJADE` package contains the following files:

- `JadeShell42P.java` and `JadeShell42PGui.java` represent a tuProlog agent and, as the name suggests, behave as a general "agent shell" for a tuProlog agent in JADE that incorporates a Prolog inference engine. When launched

in a JADE platform, the tuProlog agent needs an input file containing the Prolog theory defining the agent behaviour. The input file may be either supplied from the command line (`JadeShell42P.java`), or by browsing the file system by means of a GUI (`JadeShell42PGui.java`).
- `TuJadeLibrary.java` is a Java library necessary for a tuProlog agent to communicate in a JADE platform. It defines the communicative predicates based on the facilities that JADE offers to its agents for communication in a platform and with other platforms.

Once a `JadeShell42P` or `JadeShell42PGui` agent has been loaded into a JADE platform, it looks for the tuProlog file that contains the agents's theory. This theory must define a "`main`" predicate that implements the agent's behaviour. If this check ends positively, the tuProlog engine is created and, by default, it contains the standard tuProlog libraries and the theory input when loaded.

Every time that this agent is scheduled by JADE it automatically proves the "`main.`" goal. If the resolution does not succeed then an error message is displayed to the user.

The built-in predicates of tuProlog agents defined in the `TuJadeLibrary` include a `send(Performative,Content,Receiver,Protocol,Cid)` predicate, together with a blocking receive (`blocking_receive(Performative, Content,Sender)`) and a not blocking receive (`receive(Performative, Content,Sender)`).

Once a `th` file containing the tuProlog theory `th` for an `ag` agent has been defined, `ag` can be loaded into a JADE platform by typing `java jade.Boot ag: tuPInJADE.JadeShell42P(th)` from a command line (or `java jade.Boot ag:tuPInJADE.JadeShell42PGui`, for taking advantage of a GUI for browsing the file system).

4 Applications

The most recent application that we have developed with DCaseLP, described by [23], deals with an electronic implementation of different auction mechanisms.

There are many different auction mechanisms that can be classified according to their features (see for example [19]). The first distinction can be made between *open* and *sealed-bid* auctions. In the *open* auction mechanisms, the seller announces prices or the bidders call out the prices themselves, thus it is possible for each agent to observe the opponents' moves. The most common type of auctions in this class is the *ascending* (or *English*) auction, where the price is successively raised until no one bids anymore and the last bidder wins the object at the last price offered. The *descending* (or *Dutch*) auction, works in the opposite way w.r.t. the English one, and essentially belongs to the *sealed-bid* class. The *sealed-bid* auction mechanism is characterised by the fact that offers are only known to the respective bidders (as the name suggest, offers are submitted in sealed envelopes). In the *first-price* sealed-bid auction each bidder independently submits a single bid without knowing the others' bid, and the object is sold to the bidder who made the best offer. The *second-price* sealed-bid auction works exactly as the first-price one except that the winner pays the second highest bid.

Considering that the Dutch auction mechanisms is completely equivalent to the first-price sealed-bid auction, we have implemented the remaining three standard mechanisms: English, first-price sealed-bid and second-price sealed-bid mechanism.

Following the steps sketched in Section 2, for each auction mechanism, we have analysed the interaction between the Auctioneer role and the Bidder role, and a Protocol Diagram has been produced. In the design phase, the internal behavior and the customisable features of each class of agent has been studied. Finally, each agent has been implemented as a tuProlog agent, and integrated into DCaseLP by exploiting the functionalities offered by the `tuPInJADE` package, thus achieving the goal of providing a tuProlog library of customisable agents for simulating auction mechanisms.

We have ran all the implemented mechanisms under the hypotheses, that, according to the "Revenue Equivalence Theorem" (RET, described by [26]), lead to the existence of an optimal bidder's strategy. We programmed our test bidders with these strategies and we verified that all the simulated auctions gave the same revenue to the auctioneer and the same payoff to the bidders. The fact that RET is satisfied (up to some error clearly due to discretisation) can be seen as a check for the correctness of the implementation.

The code developed as part of this application can be downloaded from `http://www.disi.unige.it/person/MascardiV/Software/DCaseLP.html`.

Many applications had also been developed using the ancestor of DCaseLP, CaseLP. For example, the Kicker project, based on a previous "freight train traffic" application by [6], was developed within the framework of the EuROPE-TRIS Project as a result of an industrial collaboration with the Information Systems Division of Italian Railways (Ferrovie dello Stato s.p.a.), and dealt with the train dispatching problem.

Another application of CaseLP was the design and development of a working prototype of a vehicle monitoring system, which was carried out in collaboration with Elsag s.p.a. and discussed by [1].

Finally, a prototype of a multimedia, multichannel, personalised news provider (by [7]) was developed in collaboration with Ksolutions s.p.a. as part of the ClickWorld project, a research project partially funded by the Italian Ministero dell'Istruzione, dell'Università e della Ricerca (MIUR).

The above mentioned applications demonstrated that the CaseLP environment could be used effectively to engineer a real application modelled as a MAS in very heterogeneous domains.

We are currently working on making all these applications compliant with DCaseLP. Since CaseLP is implemented in Sicstus Prolog, and DCaseLP integrates tuProlog, the syntactic differences between these two Prolog implementations prevent us from running the applications developed with CaseLP in DCaseLP "as they are". However, the conversion from the two Prolog formats should be almost easy, and we plan to test soon DCaseLP on the applications already developed with its ancestor.

5 Related work

The three main features that characterise DCaseLP are: support to MAS development; support to multi-language development (at any level); support to the AOSE process. In

this section we analyse six toolkits that provide a good support to most of these features, and we compare them with DCaseLP.

AgentTool and AgentTool III. AgentTool [8] is a Java-based graphical development environment created by the Multiagent & Cooperative Robotics Laboratory. Currently, there is an ongoing project for releasing AgentTool III (aT3, `http://macr.cis.ksu.edu/projects/agentTool/agentool3.htm`) to support the design of MASs. aT3 will be released as an Eclipse plug-in, and will provide predictive performance metrics to allow the designer to make intelligent tradeoffs. It will also generate code for FIPA compliant frameworks. The *support to MAS development* provided by AgentTool consists of a set of editors that allow the developer to define the high-level system behavior in a graphical way: the types of agents as well as the possible communications that may take place between agents may be defined via the editors. This system-level specification is then refined for each type of agent in the system. Once the system has been completely specified, skeletal Java code with empty methods is produced. Once completed by hand by the developer, the Java agents may be run as any Java application. No ad-hoc monitoring and debugging facilities are provided. A good *support to multi-language specification* is provided by AgentTool: in fact, it allows the MAS developer to describe its MAS in a graphical way, providing an interface for specifying the goal hierarchy, use cases, sequence, role and agent diagrams. AgentTool allows to use different graphical languages for different specification stages. However, the language in which the high-level specifications are translated into executable code is only one: Java. As far as *support to AOSE* is concerned, AgentTool supports the Multiagent Systems Engineering (MaSE) methodology [9] that consists of capturing goals; refining roles; creating agent classes; constructing conversations; assembling agent classes; designing the system.

The INGENIAS Development Kit (IDK). The INGENIAS Development Kit (IDK), `http://ingenias.sourceforge.net/`, is a tool *supporting MAS development* thanks to the availability of the INGENIAS Editor, the main development tool for INGENIAS methodology. The editor is the replacement of Rational Rose or other UML based tools for those researchers that work with software agents, and supports alpha version of AUML protocol diagrams. As far as *support to multi-language development* in INGENIAS Development Kit is concerned, the INGENIAS Editor includes several code generation modules, among which the JADE protocol generator, that generates JADE agents that implement protocols defined with INGENIAS diagrams, and the Prolog generator, a basic, non complete, translation of INGENIAS elements to Prolog predicates. The *support to AOSE* in INGENIAS Development Kit is ensured by its adherence to the INGENIAS MAS design methodology defined by [14]. INGENIAS describes the elements that constitute a MAS, according to five viewpoints: organization, agent, goals/tasks, interactions, and environment.

The Jack Platform. JACK [24] is an agent-oriented development environment created by the Agent Oriented Software Pty Ltd, and conceived to be an environment for creating, running and integrating commercial Java-based multi-agent software using a component-based approach. JACK *supports MAS development* by supplying a lightweight implementation of the BDI architecture. Moreover, JACK provides the core architecture and infrastructure for developing and running software agents in distributed

applications, and a JDE (JACK Development Environment) that offers a high-level design tool, a graphical plan editor and graphical tracing of plan execution, that provide a powerful and flexible program development environment. JACK *does not support to multi-language development*: Java is the only language provided to implement both the agents' knowledge and their behaviour. MAS development in JACK *does not follow a principled AOSE methodology*, although the BDI approach offers a way to verify and validate the model of the application.

MadKit. MadKit [15] is a highly customisable, scalable, generic multi-agent distributed platform for developing and executing distributed applications. MadKit *supports MAS development* by providing a set of tools which are useful to the developer of multi-agent applications, like the system agents, that are the main tools that a MadKit developer uses to explore, launch, visualise and trace agents; the communicator, that is an agent which allows to build distributed applications without being concerned about distribution; and an editor and animator of diagrams that can be used to view and manipulate information represented as graphs. A "graphic shell" launches the kernel and loads the interfaces for the various agents managing them in a global GUI. MadKit provides a *good support to multi-language development*. It is possible to program MadKit agents in several languages: Java, Python, Scheme (Kawa), BeanShell and JESS. Even if MadKit *does not follow any specific AOSE approach*, one of the software tools it provides is SEdit that stands for Structure Editor. This tool allows the design and animation of structured diagrams containing nodes and arrows between them, and helps the MAS developer in engineering the MAS in a correct way.

The Mozart Programming System. The Mozart Programming System [25] is an advanced development platform for intelligent, distributed applications. It implements Oz 3, the latest in the Oz family of multi-paradigm languages based on the concurrent constraint model. By combining concurrent and distributed programming with logical constraint-based inference, Oz is *suitable for MAS development*. The developer implementing distribution must not be concerned with details regarding the underlying network, that is open and fault-tolerant. Besides this, Oz is a multiparadigm high-level programming language which supports declarative programming, object oriented programming, constraint programming, and concurrency. Thus Mozart, being based upon Oz, *provides a true support to multi-language development*. However, graphical tools for modelling the MAS or for animating diagrams describing the architecture of the MAS are not supported by the Mozart platform: *no support to any SE methodology is provided*.

The ZEUS Platform. Zeus [21] is an open source agent development tool kit written in Java and created as part of the Midas and Agentcities research projects at BT in the late 1990's and early 2000's. A version of Zeus is available under an open source license from `http://sourceforge.net/projects/zeusagent`. As far as *support to MAS development* is concerned, Zeus provides editors for entering the specifications of all the artifacts needed for building a MAS. In particular, it provides an Ontology Editor for specifying the ontology used by the agents in the MAS, and an Agent Editor for specifying agents and their tasks, social context, and social abilities. The Code Generation Editor allows the developer to automatically generate code from the specifications entered by means of the Agent Editor. Visualiser and deployment tools use

user-friendly graphic interfaces that facilitate the MAS deployment. *Multi-language issues* are not faced by Zeus: Zeus agents are programmed by entering their characterizing features through the Agent Editor panel by means of forms that impose the usage of a Zeus-dependent input language. The Java agent code can be automatically generated when all these features have been defined. No other output languages besides Java are supported. Finally, as far as *AOSE support* is concerned, the approach that Zeus suggests for building a MAS consists of five stages: ontology creation, agent creation, utility agent configuration, task agent configuration, and agent implementation.

Comparison. By comparing DCaseLP with these six toolkits, we may observe that all the seven systems provide a good support to the MAS development stage (AgentTool provides a strong support to the analysis and design stages, but poor support to the deployment and execution of the MAS, while the other tools cover all the development phases). The support that DCaseLP offers to this stage is not an original contribution, since it entirely relies on the support offered by the JADE platform, which is similar to that offered by the six systems we have analysed (apart from AgentTool). The advantage of using JADE is that it is FIPA-compliant.

The multi-language development feature is very well supported by MadKit, and fairly well supported by INGENIAS Development Kit. Instead, JACK, AgentTool, and Zeus do not offer facilities for integrating agents written in languages different from the their respective agent implementation language. Mozart allows the developer to program agents using Oz 3, that offers multi-paradigm features. Obviously, developer must know Oz 3 for programming his/her MAS.

Finally, the toolkits that better face the engineering of the MAS are AgentTool and INGENIAS Development Kit, both built for supporting an existing AOSE methodology (Mase and INGENIAS, respectively). MadKit allows the developer to define diagrams that can be animated by integrating user-defined Java code, while JACK, Mozart and Zeus offer some guidelines and tools, but no AOSE support at all.

6 Conclusions and future work

From the analysis of AgentTool, INGENIAS Development Kit, JACK, MadKit, Mozart, and Zeus, we conclude that DCaseLP is comparable with these toolkits under most respects. An advantage of DCaseLP is that it integrates a Prolog engine, which is not supported by any of the other toolkits apart from INGENIAS Development Kit. However, while INGENIAS Development Kit provides the means for generating basic, non complete Prolog predicates from INGENIAS elements, DCaseLP provides a seamless integration of Prolog agents within the JADE platform. On the other hand, a feature that is currently missing in DCaseLP is a unifying formal semantics of the agents and the MAS, despite the language they are modelled or implemented in. In the end, DCaseLP complements the related work by taking into account Logic Programming languages, that is considered in a limited way only by one of the mentioned toolkits. The advantages of exploiting Logic Programming for implementing intelligent software agents have been depicted in the Introduction of this paper.

It is part of our future work to formally describe the meaning of protocol, architecture and agent diagrams, and their relationships with the generated JESS code. We are also working on the automatic translation of all of these diagrams into tuProlog (advances on the translation of protocol diagrams has been done as part of the "West2East" project, `http://www.disi.unige.it/person/MascardiV/Software/WEST2EAST.html`, but no translation of architecture and agent diagrams has been performed yet), and on the definition of a translation program that takes as its input the XMI representations of diagrams produced by Poseidon, instead of those produced by ArgoUML.

Another direction of our research involves the integration of ontologies within DCaseLP, and the experimentation of its suitability as a tool for prototyping Service-Oriented systems. This last activity is carried out within the "Iniziativa Software Finmeccanica" project, `http://www.iniziativasoftware.it/`. Finmeccanica is the main Italian industrial group operating globally in the aerospace, defence and security sectors. The "Iniziativa Software", set up in April, 2006, is a network of public-private laboratories where researchers from both the academia and Finmeccanica, work together for applying the results obtained from the academic partners, to the industrial needs. We will exploit the results obtained in collaboration with Finmeccanica, for the industrial project within the Sistemi Intelligenti Integrati Tecnologie (S.I.I.T.) society, a non-profit consortium aimed at promoting the development of a technological district in the Italian region of Liguria, in the field of integrated intelligent systems.

References

1. E. Appiani, M. Martelli, and V. Mascardi. A multi-agent approach to vehicle monitoring in motorway. Technical report, Computer Science Department of Genova University, 2000. DISI TR-00-13, Poster session of the Second European Workshop on Advanced Video-Based Surveillance Systems, AVBS 2001.
2. F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.
3. F. Bergenti and A. Poggi. Exploiting UML in the design of multi-agent systems. In A. Omicini, R. Tolksdorf, and F. Zambonelli, editors, *Engineering Societies in the Agents World*, pages 106–113. Springer-Verlag, 2000. LNCS 1972.
4. M. J. Carr, S. L. Konda, I. Monarch, F. C. Ulrich, and C. F. Walker. Taxonomy-based risk identification. Technical report, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1993. CMU/SEI-93-TR-6 ESC-TR-93-183.
5. J. Collins and D. Ndumu. ZEUS methodology documentation, Part I: The role modelling guide. Downloadable from `http://more.btexact.com/projects/agents/zeus/`, 1999.
6. A. Cuppari, P. L. Guida, M. Martelli, V. Mascardi, and F. Zini. An Agent-Based Prototype for Freight Trains Traffic Management. In P. G. Larsen, editor, *Proceedings of the Fifth FMERail Workshop. Held in conjunction with FM'99*. Springer-Verlag, 1999.
7. M. Delato, A. Martelli, M. Martelli, V. Mascardi, and A. Verri. A multimedia, multichannel and personalized news provider. In G. Ventre and R. Canonico, editors, *Proceedings of the First International Workshop on Multimedia Interactive Protocols and Systems, MIPS 2003*, pages 388–399. Springer-Verlag, 2003. LNCS 2899.

8. S. A. DeLoach and M. F. Wood. Developing multiagent systems with AgentTool. In C. Castelfranchi and Y. Lespérance, editors, *Agent Theories Architectures and Languages, 7th International Workshop, ATAL 2000, Proceedings*, volume 1986 of *LNCS*, pages 46–60. Springer, 2000.
9. S. A. DeLoach, M. F. Wood, and C. H. Sparkman. Multiagent systems engineering. *Int. J. of Software Engineering and Knowledge Engineering*, 11(3):231–258, 2001.
10. E. Denti, A. Omicini, and A. Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Sci. Comput. Program.*, 57(2):217–250, 2005.
11. C. Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.
12. Foundation for Intelligent Physical Agents. FIPA agent software integration specification. Experimental, 15-08-2001. Downloadable from `http://www.fipa.org/specs/fipa00079/`., 2001.
13. E. Friedman-Hill. *Jess in Action : Java Rule-Based Systems (In Action series)*. Manning Publications, 2002.
14. J. Gomez-Sanz and J. Pavon. Agent oriented software engineering with INGENIAS. In V. Marík, J. P. Müller, and M. Pechoucek, editors, *3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003, Proceedings*, volume 2691 of *LNCS*, pages 394–403. Springer, 2003.
15. O. Gutknecht and J. Ferber. MadKit: a generic multi-agent platform. In *AGENTS '00: Proceedings of the fourth international conference on Autonomous agents*, pages 78–79, Barcelona, Spain, 2000. ACM Press. Home Page: `http://www.madkit.org/`.
16. B. Henderson-Sellers. Evaluating the feasibility of method engineering for the creation of agent-oriented methodologies. In M. Pechoucek, P. Petta, and L. Zsolt Varga, editors, *4th International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2005, Proceedings*, volume 3690 of *LNCS*, pages 142–152. Springer, 2005.
17. T. Juan, M. Martelli, V. Mascardi, and L. Sterling. Creating and reusing AOSE features. `http://www.cs.mu.oz.au/~tlj/CreatingAOSEFeatures.pdf`, 2003.
18. T. Juan, M. Martelli, V. Mascardi, and L. Sterling. Customizing AOSE methodologies by reusing AOSE features. In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, editors, *Proceedings of the Second International Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, pages 113–120. ACM Press, 2003.
19. P. Klemperer. *Auctions: Theory and practice*. Princeton University Press, 2004.
20. V. Mascardi, M. Martelli, and L. Sterling. Logic-based specification languages for intelligent software agents. *TPLP*, 4(4):429–494, 2004.
21. H. S. Nwana, D. T. Ndumu, L. C. Lee, and J. C. Collis. ZEUS: A toolkit for building distributed multiagent systems. *Applied Artificial Intelligence*, 13(1-2):129–185, 1999.
22. J. Odell, H. V. D. Parunak, and B. Bauer. Representing agent interaction protocols in UML. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering - First International Workshop, AOSE 2000*, pages 121–140. Springer-Verlag, Limerick, Ireland, 2000. LNCS 1957.
23. D. Roggero, F. Patrone, and V. Mascardi. Designing and implementing electronic auctions in a multiagent system environment. In *Proceedings of the WOA 2005, Dagli Oggetti Agli Agenti*. 2005.
24. The JACK Home Page. *The Agent Oriented Software Group*, 2006. Home Page: `http://www.agent-software.com/shared/home/index.html`.
25. The Mozart Home Page. *The Mozart Programming System*. Last release: June 15, 2006. Home Page: `http://www.mozart-oz.org/`.
26. W. Vickrey. Auction and bidding games. In *Recent advances in Game Theory*, pages 15–27. Princeton University Conference, 1962.