DCaseLP: a Prototyping Environment for Multilingual 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 multilingual environment for modelling and prototyping Multi-Agent Systems (MASs). DCaseLP provides an Agent-Oriented Software Engineering (AOSE) methodology which guides the MAS developer from the late requirement analysis stage to the prototype validation stage and offers a set of languages and tools both for modelling agents, and for developing a prototype of the MAS in a semi-automatic way. Full support for validating the MAS model by running the prototype in a JADE platform is offered. DCaseLP has been used to develop an e-commerce application, thus demonstrating the advantages of rapid prototyping in AOSE.

Keywords. Multi-Agent System, Multilinguality, Agent-Oriented Software Engineering, Rapid Prototyping.

1 Introduction

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). The lack of mature methodologies, tools, and environments for agent-based systems development limits the effectiveness and impact of AOSE, as reported by AgentLink III (2004). Researchers and practitioners agree that engineering a multi-agent system (MAS) still involves a non negligible degree of risk, which we feel could be lowered by using a prototyping approach to MAS engineering, because:

- 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.
- The cost of the software prototype is usually much lower than the cost of the end product, and this reduces the financial risks involved in the development process.

- The iterative prototyping process allows us the flexibility to revise the requirements or critical design choices several times before committing to any final decision.
- 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, but not necessarily for the implementation of the final product.

In this paper we describe the DCaseLP framework. 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 multilingual prototyping environment that integrates both imperative (object-oriented) and declarative (rule-based and logic-based) languages, as well as graphical ones (UML, `http://www.uml.org/`). 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 language to model, implement, or test each property, DCaseLP heads towards a modular approach to AOSE proposed by Juan et al. (2003a,b). 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.

The paper is organised in the following way: Section 2 discusses the AOSE stages that DCaseLP addresses; Section 3 describes the tool from a user perspective; Section 4 discusses the most recent application of DCaseLP. Section 5 compares DCaseLP with relevant related tools, and concludes.

2 DCaseLP: an Integrated AOSE Methodology and Environment

In order to quickly develop, following a principled approach, a MAS prototype, DCaseLP supplies a development methodology that covers all the engineering stages from late requirements analysis to prototype testing. Early requirements analysis is not included in the method and it can be handled by providing a natural language description of the prototype goals, functionalities, and application domain.

The tools and languages that DCaseLP offers 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.

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

2.1 Modelling stage (analysis and design)

The analysis stage is mainly role-driven. Our belief, that roles are the key abstraction in MAS modelling, is shared by several researchers in the AOSE field (for example, Cabri et al. (2002); Kendall (2000); Zambonelli et al. (2003); Zambonelli and Omicini (2004)). Role modelling allows us to specify *what* the system can do, without going into the details about *how* the system will do it.
According to Collins and Ndumu (1999), the qualifying criteria for roles are:
– *Modularity*: a role describes a set of entities that can occupy the same position in a re-occurring structure, hence a role must be modular so that new players can be assigned without any impact on the rest of the role model.
– *High cohesion*: to promote modularity a role must have a well-defined set of related responsibilities and a clearly defined function; the responsibilities must form a cohesive unit.
– *Parsimony*: a role should not have extraneous responsibilities.
– *Completeness*: a role should not be trivial; trivial roles should be merged with other roles.
– *Low coupling*: dependency among roles must be minimised.

Since we agree with this characterisation of roles, we suggest that the MAS developer should define the roles to be played within the MAS and the interactions taking place among roles on the basis of the above mentioned guidelines.

The language that DCaseLP provides to the user in order to face this engineering stage is UML, along the lines of Bergenti and Poggi (2000) and Collins and Ndumu (1999). The tools that allow the integration of role models defined using UML into DCaseLP are a set of XSL configuration files that define the rules for translating XMI representations of UML diagrams into executable code (see Figure 1).

Once the role model is well understood, the developer needs to address the following:

1. which roles to assign to each agent class (not a Java class!) in the architecture diagram;

2. the number of instances of each agent class (in the agent diagram) that is required for the given application.

In order to assign roles to an agent class, the developer may follow, for example, the two criteria suggested by Collins and Ndumu (1999):

1. the sphere of responsibility test and

2. the point of interaction test,

although, in our model, roles are associated to an agent class and not to an agent instance.

The sphere of responsibility test derives from the requirement that agents must be *autonomous*, i.e., be responsible for the control of resources and provision of services. This area of control is known as the agent's *sphere of responsibility*. Thus, when taking

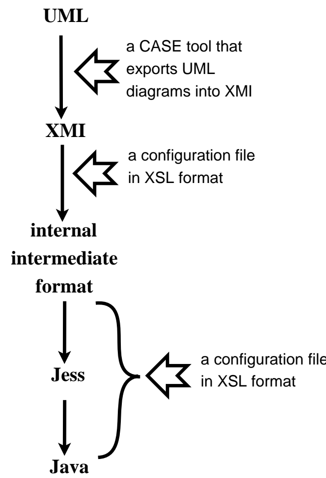

Figure 1: Translation from UML into executable code.

into consideration which classes of agents will exist, the developer needs to consider how the application domain will be partitioned, i.e., the degree of coupling among roles. The point of interaction test extends the sphere of responsibility by taking into consideration the *sociality* of agents. The purpose of this test is to help separate application resources from the entities that will use them to provide services. The test suggests that the access point for information, expertise, and services is a good agent class candidate.

The number of instances of each agent class depends on the specific system that is under development. However, clearly decoupling agent classes from agent instances enforces the modularity and re-usability of the agent class model. For each agent class defined in this stage, the corresponding code should be defined in the next stage.

DCaseLP not only provides the user with the UML language, in order to face the class modelling phase, but also a semi-automatic translation tool from UML diagrams representing agents classes to JESS code that can be integrated into JADE.

2.2 Implementation Stage

The implementation of the MAS prototype must be coherent with the specifications mentioned in the previous steps. Ensuring this coherence is a demanding task for the developer of the prototype, but DCaseLP reduces 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. External software integration in DCaseLP follows the *Wrapper Agent* model defined by the Foundation for Intelligent Physical Agents (2001), 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.

We should remark, here, that we provide a set of tools 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 these agents 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 us 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 these criteria:

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 whether agents 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 methodology* described in Section 2, and
2. a *transparent integration* between Java, JESS, and tuProlog agents running in a JADE platform.

The result of our work consists of:

1. the Java `UMLInJADE` package that contains the Java classes and the XSL style sheets that allow the user to translate UML diagrams (created with the ArgoUML modelling tool, 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.
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 "readme" files, manuals and short tutorials, can be downloaded from `http://www.disi.unige.it/person/MascardiV/Software/DCaseLP.html`.

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, given either in UML or in an XML intermediate format, into partial JESS agents. These agents will become complete in the final stage, when the developer inserts the details regarding the specific application for which the MAS is created.

The protocol diagrams set 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 the XML intermediate format that we have developed. 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 sequence diagrams (and protocol diagrams are basically sequence diagrams). The current version of Poseidon (`http://www.gentleware.com/index.php`), that supports the definition of sequence diagrams, exports 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.

To show what our intermediate XML is like, we use it, for example, to describe the propose protocol suggested by FIPA (shown in Figure 2); the resulting intermediate XML specification is the following (we assume that a Seller role substitutes the role of Initiator, and a Buyer role substitutes the role of Participant):

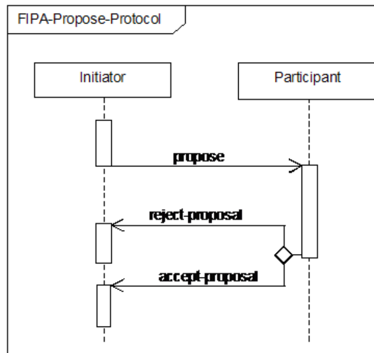

Figure 2: FIPA propose protocol.

```
<protocoldiagram>
   <role><name>Seller</name>
       <msgs><msg><sender>Seller</sender>
           <receiver>Buyer</receiver>
           <act>PROPOSE</act></msg>
       <xor-thread>
         <thread><msg>
           <sender>Buyer</sender><receiver>Seller</receiver>
           <act>REJECT_PROPOSAL</act></msg></thread>
         <thread><msg>
           <sender>Buyer</sender><receiver>Seller</receiver>
           <act>ACCEPT_PROPOSAL</act></msg></thread>
       </xor-thread></msgs></role>
   <role><name>Buyer</name>
       <msgs><msg>
          <sender>Seller</sender><receiver>Buyer</receiver>
          <act>PROPOSE</act></msg>
       <xor-send>
          <sender>Buyer</sender><receiver>Seller</receiver>
          <act>REJECT_PROPOSAL</act>
          <act>ACCEPT_PROPOSAL</act>
       </xor-send></msgs></role>
</protocoldiagram>
```

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

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

The `UMLInJADE` package 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, 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

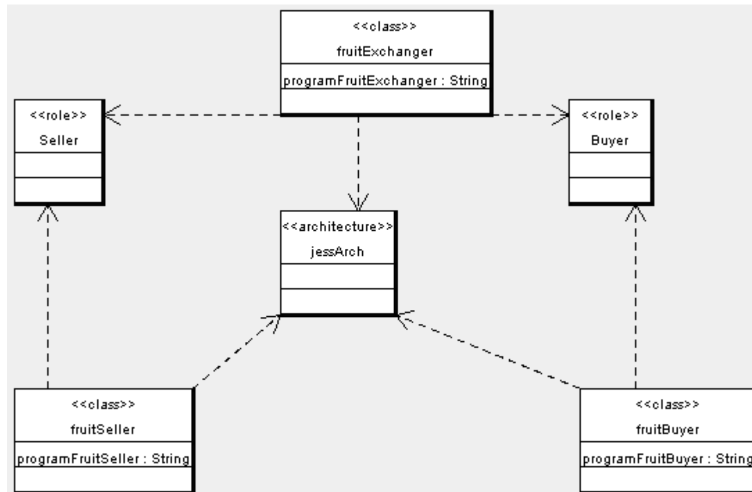

Figure 3: An architecture diagram.

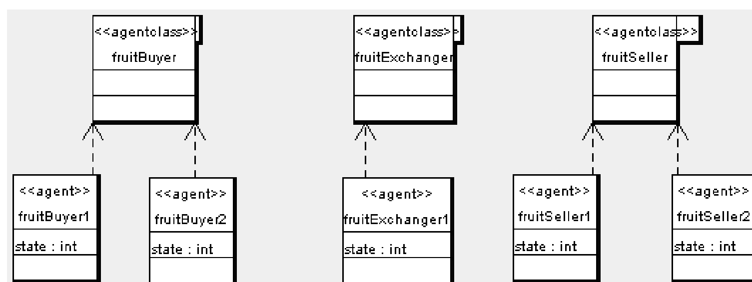

Figure 4: An agent diagram.

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 files, that is required by means of a set of interactive windows. The output of the translation program is put in the jacode (for the Java code) and jecode (for the JESS agents) directories, in the UMLInJADE directory.

In order to execute the MAS resulting from the translation of the high level specification, the generated JESS code must first be completed. The JESS code is characterised by an initial part that defines the functions available to the JESS agent, followed by the rules that implement the interaction protocol defined when specifying the MAS. The developer must complete these rules where indicated by the JESS code itself, by means of self-explaining comments.

Once all the JESS agents have been completed, the JADE MAS can be built and its simulation can start. First of all, the Java pieces of code must be compiled in order to obtain the Java classes corresponding to the agent classes. This can be done by typing javac*.java from the jacode directory. Supposing that the MAS agent diagram is the one specified in Figure 4, then, always from the jacode directory, the MAS simulation in JADE can be started by typing java jade.Boot fB1:fruitBuyer fB2:fruitBuyer fS1:fruitSeller fS2:fruitSeller fE1:fruitExchanger (we have substituted fruitBuyer1, etc, with fB1, etc, for readability).

3.2 The jessInJADE package

The jessInJADE package defines two classes, jessAg and jessBhv. The most important among them is the jessAg class, since every JESS agent must be defined by means of a Java class that extends jessAg. The jessAg class, in turn, extends the JADE Agent class by adding to it the capability to exchange messages with any JADE agent, while the class jessBhv defines the behaviour of a JESS agent, which is a cyclic behaviour.

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, that 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), and saved with the name chosen for this agent class, 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.

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. The arguments of the slots must be either a variable or a string. The performative of the message must be specified in capital letters, the receiver must be specified by a string (or many strings separated

by blank spaces) representing the ID of an agent in JADE. 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:

- `ErrorMsg.java`: it defines the Java class `ErrorMsg` that is used by the tuProlog agents with the aim of displaying a pop-up window with error and failure messages, since the JADE platform does not provide its agents with a similar mechanism.

- `JadeShell42P.java`: this represents a tuProlog agent and, as the name suggests, it behaves as a general "agent shell" for a tuProlog agent in JADE that incorporates a Prolog inference engine (implemented by tuProlog) and, when launched in a JADE platform, needs an input file containing the Prolog theory defining the agent behaviour.

- `JadeShell42PGui.java`: is similar to the `JadeShell42P` class, but differs from it since when loading the agent it displays an additional GUI from which the user can browse the file system and select the theory file to input to the agent.

- `TuJadeLibrary.java`: it is a tuProlog library (developed entirely in Java) necessary for a tuProlog agent to communicate in a JADE platform since 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, its first action is to check if the user has inputted the name of the file that defines the "`main`" predicate; if this check ends positively, the tuProlog engine is created and, by default, it contains the standard tuProlog libraries and the theory inputted when loaded.

After adding the inference engine, the agent tries to extend it with the `tuPInJADE.TuJadeLibrary`: if it succeeds, the agent can now communicate with any agent running in JADE platforms, otherwise it terminates its life cycle. 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 tuProlog theory `th` that defines the behaviour of a tuProlog agent `ag` has been defined, the agent `ag` can be loaded into a JADE platform by typing `java jade.Boot ag:tuPInJADE.JadeShell42P(th)` from a command line.

4 Applications

The most recent application that we have developed with DCaseLP, described by Roggero et al. (2005), 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 Klemperer (2004); Krishna (2002)). 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 DCaseLP methodology 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 Vickrey (1962); Myerson (1981); Riley and Samuelson (1981)), 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 Cuppari et al. (1999), 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 Appiani et al. (2000).

Finally, a prototype of a multimedia, multichannel, personalised news provider (by Delato et al. (2003)) 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 and conclusions

The three main features that characterise DCaseLP are:

1. support to MAS development;
2. support to multilinguality (at any level);
3. support to the AOSE process.

Three toolkits that provide a good support to most of these features, and thus that can be compared with DCaseLP, are MadKit (a **M**ulti-**A**gent **D**evelopment **K**it) by Gutknecht and Ferber (2000), the Mozart Programming System by The Mozart Consortium (2005), and JACK (**J**ava **A**gent **C**ompiler and **K**ernel) (see The JACK Home Page (2005)). The reader can refer to Gungui (2004) for a deep comparison of DCaseLP with these three toolkits, whose results are here just summarised.

By comparing DCaseLP with these three toolkits, we can conclude that all the four systems provide a good support to the MAS development stage. 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 three systems we have analysed. The advantage of using JADE is that it is the only FIPA-compliant platform, among those analysed in this section.

The multilinguality feature is very well supported by MadKit, that allows the developer to program agents in Java, Python, Scheme (Kawa), BeanShell and JESS, while JACK does not offer facilities for integrating agents written in languages different from the its proprietary agent language. Mozart allows the developer to program agents using Oz 3, the latest in the Oz family of multi-paradigm languages based on the concurrent constraint model. Thus, a support to multilinguality is, in some sense, provided by the multi-paradigm features of Oz 3, but the developer must know Oz 3 for programming his/her MAS.

Finally, the only toolkit that seems to face the engineering of the MAS in a principled way is MadKit, that allows the developer to define diagrams that can be animated by integrating user-defined Java code. Both Mozart and JACK offer some guidelines and tools, but no AOSE methodology at all.

From the analysis of these three toolkits, a feature that is currently missing in DCaseLP, and that is instead provided by other toolkits, emerged: a unifying formal semantics of the agents and the MAS, despite to the language they are modelled or implemented in. 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 these diagrams into tuProlog, 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. Finally, we plan to test DCaseLP on the applications that we successfully developed using CaseLP.

References

AgentLink III, 2004. Agent technology roadmap: Overview and consultation report. www.agentlink.org/roadmap/roadmapreport.pdf.

Appiani, E., Martelli, M., Mascardi, V., 2000. A multi-agent approach to vehicle monitoring in motorway. Tech. rep., Computer Science Department of Genova University, dISI TR-00-13, Poster session of the Second European Workshop on Advanced Video-Based Surveillance Systems, AVBS 2001.

Bergenti, F., Poggi, A., 2000. Exploiting UML in the design of multi-agent systems. In: Omicini, A., Tolksdorf, R., Zambonelli, F. (Eds.), Engineering Societies in the Agents World. Springer-Verlag, pp. 106–113, lNCS 1972.

Cabri, G., Leonardi, L., Zambonelli, F., 2002. Modeling role-based interactions for agents. In: Debenham, J., Henderson-Sellers, B., Jennings, N., Odell, J. (Eds.), Proceedings of the OOPSLA'02 Workshop on Agent-Oriented Methodologies.

Collins, J., Ndumu, D., 1999. ZEUS methodology documentation, Part I: The role modelling guide, downloadable from http://more.btexact.com/projects/agents/zeus/.

Cuppari, A., Guida, P. L., Martelli, M., Mascardi, V., Zini, F., 1999. An Agent-Based Prototype for Freight Trains Traffic Management. In: Larsen, P. G. (Ed.), Proceedings of the Fifth FMERail Workshop. Held in conjunction with FM'99. Springer-Verlag.

Delato, M., Martelli, A., Martelli, M., Mascardi, V., Verri, A., 2003. A multimedia, multichannel and personalized news provider. In: Ventre, G., Canonico, R. (Eds.), Proceedings of the First International Workshop on Multimedia Interactive Protocols and Systems, MIPS 2003. Springer-Verlag, pp. 388–399, lNCS 2899.

Foundation for Intelligent Physical Agents, 2001. FIPA agent software integration specification, experimental, 15-08-2001. Downloadable from http://www.fipa.org/specs/fipa00079/.

Gungui, I., 2004. Integrazione di agenti logici in DCaseLP. Master's thesis, Computer Science Department of Genova University, Italy, in English.

Gutknecht, O., Ferber, J., 2000. MadKit: a generic multi-agent platform. In: AGENTS '00: Proceedings of the fourth international conference on Autonomous agents. ACM Press, Barcelona, Spain, pp. 78–79, home Page: http://www.madkit.org/.

Juan, T., Martelli, M., Mascardi, V., Sterling, L., 2003a. Creating and reusing AOSE features, http://www.cs.mu.oz.au/~tlj/CreatingAOSEFeatures.pdf.

Juan, T., Martelli, M., Mascardi, V., Sterling, L., 2003b. Customizing AOSE methodologies by reusing AOSE features. In: Rosenschein, J. S., Sandholm, T., Wooldridge, M., Yokoo, M. (Eds.), Proceedings of the Second International Conference on Autonomous Agents and Multiagent Systems (AAMAS'03). ACM Press, pp. 113–120.

Kendall, E. A., 2000. Role modelling for agent systems analysis, design and implementation. IEEE Concurrency 2 (8), 34–41.

Klemperer, P., 2004. Auctions: Theory and practice. Princeton University Press.

Krishna, V., 2002. Auction Theory. Academic Press.

Myerson, R., 1981. Optimal auction design. Mathematics of Operations Research 6, 58–73.

Riley, J., Samuelson, W., 1981. Optimal auctions. American economic review 71, 381–92.

Roggero, D., Patrone, F., Mascardi, V., 2005. Designing and implementing electronic auctions in a multiagent system environment. In: Proceedings of the WOA 2005, Dagli Oggetti Agli Agenti.

The JACK Home Page, 2005. *The Agent Oriented Software Group*. Home Page: http://www.agent-software.com/shared/home/index.html.

The Mozart Consortium, 2005. *The Mozart Programming System*. Home Page: http://www.mozart-oz.org/.

Vickrey, W., 1962. Auction and bidding games. In: Recent advances in Game Theory. Princeton University Conference, pp. 15–27.

Zambonelli, F., Jennings, N. R., Wooldridge, M., 2003. Developing multiagent systems: The Gaia methodology. ACM Transactions on Software Engineering and Methodology 12 (3).

Zambonelli, F., Omicini, A., 2004. Challenges and research directions in agent-oriented software engineering. Autonomous Agents and Multi-Agent Sytems 9, 253–283.