

Integrating tuProlog into DCaseLP to Engineer Heterogeneous Agent Systems^{*}

Ivana Gungui and Viviana Mascardi

Dipartimento di Informatica e Scienze dell'Informazione – DISI,
Università di Genova, Via Dodecaneso 35, 16146, Genova, Italy.
1995s133@educ.disi.unige.it, mascardi@disi.unige.it

Abstract. Scrivo due righe e te le mando verso le 14; tu inseriscile QUI

1 Introduction

Multiagent Systems (MASs) involve heterogeneous components which have different ways of representing their knowledge about the world, about themselves, and about other agents, and which adopt different mechanisms for reasoning about this knowledge. Despite heterogeneity, agents need to interact and exchange information in order to cooperate or compete for the control of shared resources; this interaction may follow sophisticated communication protocols.

For these reasons and due to the complexity of agents behaviour, MASs are difficult to be correctly and efficiently engineered. Even developing a working prototype may require a long time and a lot of different skills. In fact, the prototype can involve agents that would better be modelled and implemented using a language based on Horn clauses, agents that would be easily defined using an expert system-like language, and others that should be directly implemented in some implementation language, in order to access existing software packages or the web. Moreover, some general aspects of the MAS can be better specified using ad-hoc specification languages. For example, the MAS architecture, the internal agent architecture and the interaction protocols among agents can be easily specified using graphical tools and languages.

The development of a prototype system of heterogeneous agents can be carried on in different ways. A first -trivial- solution consists of developing all the agents by means of the same implementation language and to execute the obtained program. If this approach is adopted, during the specification stage it would be natural to select a specification language that can be directly executed or easily translated into code, and to specify all the agents in the MAS using it. An opposite solution would be to specify each “view” of the MAS (including the MAS architecture, the interaction protocols among agents, the internal architecture and functioning of each agent) using the most suitable language in order to deal with the MAS’s peculiar features, and to verify, execute, or animate the obtained specifications inside an integrated environment. Such an

^{*} Parts of this document appear in the manuscript “Reasoning about Communicating Agents inside DCaseLP” by M. Baldoni, C. Baroglio, I. Gungui, A. Martelli, M. Martelli, V. Mascardi, V. Patti, C. Schifanella. Submitted.

environment should offer the means to select the proper specification language for each view of the MAS, and to check the specifications. This check may be carried out thanks to formal validation and verification methods or by producing an executable code and running the prototype thus obtained.

Despite its greater complexity, the last solution has many advantages over the first, trivial one.

1. By allowing the use of different specification languages for each view of the MAS, *it supports the progressive refinement of specifications*: for example, the specification of an interaction protocol performed during the early analysis stage does not need to be as detailed as the complete specification of an agent performed during the detailed design stage; details can be progressively added while the engineering process goes on.
2. By allowing the use of different specification languages for the internal architecture and functioning of each agent, *it respects the differences existing among agents*, namely the way they reason and the way they represent their knowledge, the other agents, and the world.
3. By allowing different implementation languages to be integrated inside the same running prototype, *it allows the direct implementation of some of the agents*, skipping the specification stage.
4. In case more than one language fits the requirements of an agent/view under specification, *it allows the developer to choose the language he/she knows best and likes*, thus leading to more reliable specifications and implementations.

Currently, solid and complete environments that allow the integration of heterogeneous specification and implementation languages in a seamless way do not exist yet, but some preliminary steps have been made in this direction, and some initial results have already been achieved with the development of prototypical environments for engineering heterogeneous agents. DCaseLP (Distributed CaseLP), integrates a set of specification and implementation languages in order to model and prototype MASs and defines a methodology which covers the engineering stages from requirements analysis to prototype execution, which relies on the use of AUML both at the requirement analysis level and for describing the *interaction protocols* followed by the agents. Although the first release of DCaseLP [10,1] demonstrates that the concepts underlying the “integrated environment for engineering heterogeneous MAS” can be put into practice and can give interesting results, it suffers from two limitations that affect its applicability:

1. it does not provide the means to re-use the code and instruments already developed for the ancestor of DCaseLP, CaseLP [11]; and
2. it does not provide tools and languages for reasoning about properties of the interactions occurring among the agents.

The last limitation can be addressed by translating AUML interaction protocols into the DyLOG language for reasoning about actions and changes [7,4,6], and then integrating DyLOG into DCaseLP [5], while the first limitation can be overcome by extending DCaseLP with the ability to integrate agents specified as Prolog theories, as discussed in this paper.

The structure of the paper is the following: Section 2 overviews the DCaSeLP environment and discusses the outcomes of integrating an existing Prolog implementation, tuProlog, into DCaSeLP, while Section 3 discusses the technical details of this integration. Section 4 shows an example of use of DCaSeLP extended by tuProlog; conclusions follow.

2 The DCaSeLP environment

DCaSeLP is a prototyping environment where agents specified and implemented in a given (and fairly limited!) set of languages can be seamlessly integrated. DCaSeLP provides an agent-oriented software engineering methodology to guide the MAS developer during the analysis of the MAS requirements, the MAS design, and the development of a working MAS prototype. The methodology is sketched in Figure 1. Solid arrows represent the information flow from one step to the next one. Dotted arrows represent the iterative refinement of previous choices. The first release of DCaSeLP did not face all the stages of the methodology. In particular, the verification stage was not addressed. The integration of tuProlog into DCaSeLP discussed in Section 3 will allow us to address also the verification phase.

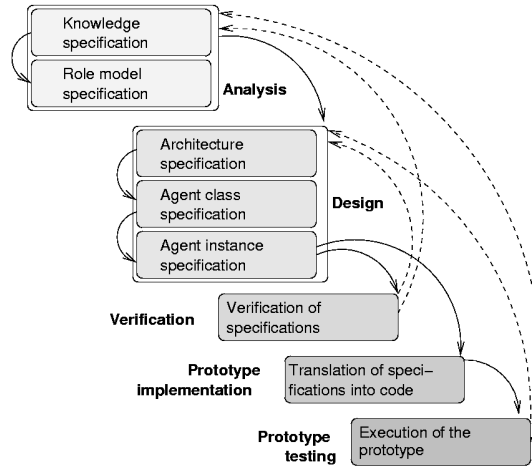


Fig. 1. DCaSeLP methodology.

DCaSeLP is the result of the effort to re-implement CaseLP [11] in order to overcome its main limitations, namely:

1. its centralization,
2. its poor support to concurrency, and
3. its lack of adherence to existing standards.

The tools and languages supported by the first release of DCaSeLP, discussed in [10,1], are represented in Figure 2 by means of the darker boxes. Lighter boxes represent the desired extensions with respect to that release. Some of these extensions have already been made, while some are currently being made, and some are just part of our future work.

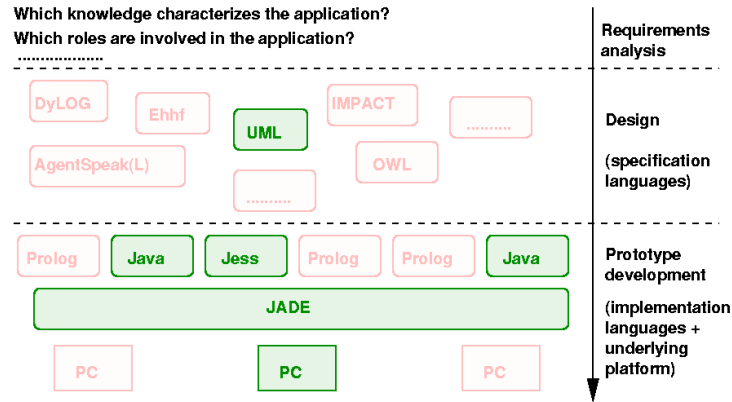


Fig. 2. Tools and languages supported by DCaSeLP, first release.

DCaSeLP adopts an existing multiview, use-case driven and UML-based method [2,3] in the phase of requirements analysis.

Once the requirements of the application have been clearly identified, the developer can use UML and its Agent-oriented extension AUMML to describe the interaction protocols followed by the agents, the general MAS architecture and the agent types and instances. Moreover, the developer can also automatically create the rule-based code for the agents in the MAS in such a way that the UML/AUMML specification is satisfied. In the following we will assume that AUMML is used during the requirements analysis stage, although the translation from AUMML into rule-based code is not fully automated (while the translation from pure UML into code is).

The rule-based language used for the implementation of DCaSeLP agents is Jess [9]. The Jess code obtained from the translation of AUMML diagrams must be manually completed by the developer with the behavioral knowledge which was not explicitly provided at the specification level. The developer does not need to have a deep insight in rule-based languages in order to complete the Jess code, since he/she is guided by comments included in the automatically generated code. In this way, a developer who is not confident with rule-based languages can concentrate on the AUMML specification and make a little effort to complete the rule-based code in order to make it executable.

On the other hand, the developer who prefers to define agents in a declarative language, can skip the AUMML specification stage and directly write the Jess code.

The choice of Jess as the language for implementing agents was lead by two considerations:

1. being a rule-based language, Jess is suitable for representing both the event-driven and the goal-driven behaviors of the agents;
2. being implemented in Java, Jess can be easily integrated into the FIPA-compliant JADE platform.

JADE (Java Agent Development Framework, [8]) is both a middle-ware that complies with the FIPA specifications [13] and a set of graphical tools that support the debugging and deployment phases. The agents can be distributed across several machines and they can run concurrently. The adoption of JADE as the underlying platform for implementing DCaseLP was a must in order to overcome the three limitations of CaseLP. In fact, JADE is distributed, allows the concurrent execution of agents, and is FIPA-compliant. By integrating Jess into JADE, we were able to easily monitor and debug the execution of Jess agents thanks to the monitoring facilities that JADE provides. The experiments carried out with the first release of DCaseLP were on a single machine (see Figure 2: there is only one dark box labelled with “PC” under the JADE box).

The possibility of running the prototype allowed the first release of DCaseLP to demonstrate its ability in checking the coherence of the AUMML diagrams produced during the requirements analysis step. Performing such a check is a well known and still open problem that we could face without additional effort. Nevertheless, that release still suffered from one limitation: it was not able to integrate any Prolog implementation. The ancestor of DCaseLP, namely CaseLP, is implemented in Sicstus Prolog [12], and a lot of work has been done to study and define semi-automatic translators from high-level specification languages into CaseLP agents, namely agents described in Sicstus Prolog extended with communication primitives. Limited support to formal verification of specifications – completely missing in DCaseLP – is indeed provided by CaseLP. Without the integration of Prolog into DCaseLP, all that work would have been lost. Recently, we have extended DCaseLP with the ability to integrate agents specified as Prolog theories. Section 3 discusses how we have integrated an existing Prolog implementation, tuProlog [14], into DCaseLP. The choice of tuProlog was due to two of its features:

1. it is implemented in Java, which makes its integration into JADE easier, and
2. it is very light, which ensures a certain level of efficiency to the prototype.

The integration of tuProlog into DCaseLP has been completed at the beginning of April 2004. Due to the syntactic differences existing between Sicstus Prolog and tuProlog, CaseLP agents specified using Sicstus Prolog cannot be simply treated as if they were DCaseLP agents specified using tuProlog: a translation step from “Sicstus Prolog for CaseLP agents” to “tuProlog for DCaseLP agents” is necessary. We guess that this translation step can be easily automatized, thus allowing us to re-use the tools developed for CaseLP inside DCaseLP, but, for lack of time, we did not start working at its implementation. Always due to lack of time, we did not verify the ability to run

JADE, Jess and tuProlog agents as part of the same, heterogeneous, MAS. At the time of writing, we have only developed some examples (one of which is discussed in Section 4) that demonstrate that tuProlog agents are able to interact with both tuProlog and JADE agents by taking advantage of the underlying communication middleware provided by JADE, and that the execution of the resulting MAS can be monitored using the tools offered by JADE. When the translator “*Sicstus Prolog* \rightarrow *tuProlog*” will be ready, and when the compatibility between Jess and tuProlog agents will be fully established, DCaseLP will be closer than now to the integrated environment for engineering heterogeneous MASs envisaged in Section 1. In particular,

1. *It will support the progressive refinement of specifications:* for example, the interactions among agents belonging to the MAS and among internal components of the same agent will be specified in some suitable language (AUMML, other languages provided by CaseLP), will be then formally verified, and will be finally implemented by adding all the details needed by the MAS or by the single agent to work.
2. *It will respect the differences existing among agents:* an agent which reasons in a goal-driven, backward fashion will be easily defined by means of a tuProlog theory; a rule-based agent will be better defined using Jess.
3. *It will allow the direct implementation of some of the agents:* JADE agent are basically Java agents and thus they are implemented agents, rather than specified agents.
4. *It will allow the developer to choose the language he/she knows best and likes:* it will provide a large bunch of languages to choose from.

3 Integrating tuProlog into DCaseLP

The integration of tuProlog into DCaseLP has been carried out in order to provide the developer of the MAS with a way to define the behavior of an agent by means of another declarative language besides Jess, and to re-use the code and instruments previously developed for CaseLP. To do so, tuProlog has been integrated into JADE.

JADE includes a specific package to develop Java agents and a programmer’s guide containing implementation guidelines that the developer should follow to code his/her agents in Java. Any Java class that extends the class `Agent` defined in the package `jade.core` of JADE can be considered as a JADE agent. To add tuProlog in DCaseLP, three Java classes have been defined in a package named `tuPInJADE`:

1. the class `JadeShell42P`, which represents a tuProlog agent in JADE;
2. the class `JadeShell42PGui` that provides an additional GUI at the loading of the agent; and
3. the class `TuJadeLibrary`, which is a tuProlog library (developed in Java) necessary to a tuProlog agent in order to communicate in the JADE platform.

As the name of the class wants to suggest, `JadeShell42P` behaves as a shell for a tuProlog engine. To execute a `JadeShell42P` agent in JADE, the programmer has to give in input the name of a file containing a tuProlog theory that represents the behavior of the agent (Figure 3). The class `JadeShell42PGui` differs from the

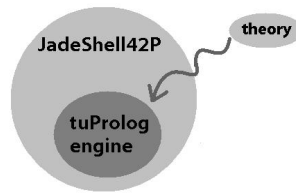


Fig. 3. JADE shell for a tuProlog engine.

class `JadeShell42P` in the fact that, when loaded in JADE, it does not need in the command line the name of the theory file: it loads the pop-up window shown in Figure 4 with which the user can browse the file system and select from the list of files the one containing a tuProlog theory to be used as behaviour of the agent. Such tuProlog

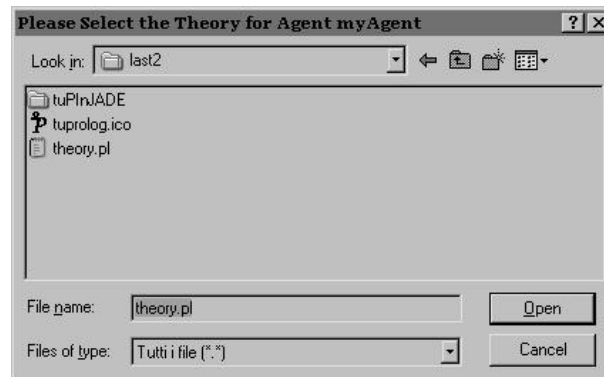


Fig. 4. Window for theory selection.

theory file has only one restriction: it has to begin with the definition of a predicate called `main/0`. When a tuProlog agent is loaded in JADE, it first creates a tuProlog engine containing the standard tuProlog libraries and then extends it by loading the ad hoc tuProlog library named `TuJadeLibrary`. The behavior of any tuProlog agent is to use the tuProlog engine, created during their initialization phase, always to solve the predicate `main`. A typical `main` predicate will call predicates to read a new message, handle it and do some actions (as the update of the agent's knowledge and message delivery) according to the previous steps.

The goal's demonstration is not visible to the programmer: if he/she wants to be informed of the variable's bindings made during the resolution, he/she has to explicitly write the variables on the standard output or in some files that he/she can subsequently go and read. The only explicit information which is provided to the user regards the failure of the goal's demonstration and other situations which raise an error during the

resolution process. To make this information visible, the package `tuPinJADE` defines the Java class `ErrorMsg` that is used by the `tuProlog` agents to pop-up a window displaying error and failure messages, like the one shown in Figure 5.

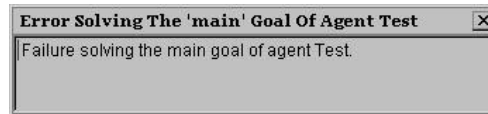


Fig. 5. Window for error and failure messages.

The Java class `JadeShell42P` defines the inner class `Shell42PBehaviour` (named `Shell42PBehaviourGui` in the class `JadeShell42PGui`) that extends the Java class `CyclicBehaviour` defined in the package `jade.core.behaviours` of JADE.

`Shell42PBehaviour` implements the only behavior of a `JadeShell42P` agent: it is executed forever, in other words, every time the agent is scheduled by the JADE's scheduler, it tries to fulfill only one activity, that is, the resolution of the goal `main`. The `Shell42PBehaviour` models a cyclic task and cannot be interrupted while executing its action method. The result is the same as if the agent performed a “while true do main” statement, with `main` being dealt with as an atomic action.

The Java class `TuJadeLibrary` is the core class regarding the communication of the `tuProlog` agents in JADE. This library defines the predicates `send` and `receive`: they are the directives implementing the sending and receiving of the FIPA compliant and asynchronous messages to and from agents of a JADE platform. The `send` and `receive` predicates simply invoke the `send` and `receive` methods of the JADE agents, therefore they allow communication among `tuProlog` agents but also among ordinary JADE agents and `tuProlog` agents.

The arguments of the `send` predicate are: the performative, the content and the JADE address/list of addresses of the receiver/receivers of the message to send. The arguments of the `receive` predicate are: the performative, the content and the JADE address of the sender of the message received. Actually, since JADE agents have the possibility to stop their activity while waiting for a message to arrive in their messages queue, the `TuJadeLibrary` also defines two `blocking_receive` predicates: one without a timeout and the other one with a timeout. These predicates correspond to the `blockingReceive` method of an ordinary JADE agent.

Finally, `TuJadeLibrary` defines two predicates for converting strings into terms and vice-versa, named `pack` and `unpack`. They allow `tuProlog` agents to send strings as the content of their messages, and to reason over them as if they were Prolog terms.

4 Example

To show how `DCaseLP` can be used to develop a working MAS prototype, we use a simple example drawn from a distributed marketplace scenario.

In our marketplace, there are two agents (buyer1 and buyer2) that want to buy some fruit (oranges, apples and kiwi) from three agents (seller, seller1 and seller2). Agents buyer1, buyer2, seller1 and seller2 are all tuProlog agents, while seller is an ordinary JADE agent.

The agents that sell fruit can receive two kind of messages from the buyers:

1. a request for price: the ACLMessage received has the performative REQUEST and the content price(Fruit), where Fruit is oranges or apples or kiwi;
2. a request for buying: the ACLMessage received has the performative REQUEST and the content buy(Fruit, Amount), where Fruit is oranges or apples or kiwi, while Amount is the quantity of fruit that the buyer wants to buy.

A seller replies to a price request made by a buyer by sending an INFORM ACLMessage that has the content price(Fruit, Price), where Fruit is oranges or apples or kiwi and Price is the corresponding price.

The reply to a request for buying depends on whether or not the seller has enough fruit to sell: in case the quantity of fruit that the buyer is willing to buy is less or equal to the one possessed by the seller, the seller will send the buyer an INFORM ACLMessage with the content bought(Fruit), to inform the buyer that the fruit Fruit has been sold. On the other hand, if the seller does not own enough fruit, it sends the buyer an INFORM ACLMessage with the content no_more(Fruit), so the buyer will know it can no longer buy Fruit from that seller.

At the beginning, the buyers send a request for the price of all the fruit to all the sellers. Once they know the prices of the fruit, they send requests for buying the fruit to the agents that sell at the cheapest price that fruit. The buyers keep sending messages requesting to buy fruit until they have money or the sellers have enough fruit to sell.

To give the flavor of how a tuProlog agent looks like, Figure 6 shows a piece of the tuProlog theory associated to buyer1. The main predicate defines three activities which consist in handling incoming messages, asking the price of the fruit to seller (this activity will be performed only once) and buying fruit. After the definition of the main, the initial state of the buyer agent is defined: buyer1 possesses no fruit, buys oranges in stocks of 2 kilos, apples in stocks of 3 kilos and kiwi in stocks of 12 kilos, and has 200 Euro to spend. The list of addresses of the seller agents follow (sellers_addresses([...])), together with other information that we do not show for sake of conciseness.

Handling messages consists of receiving one of them (call to the receive predicate provided by the TuJadeLibrary and introduced in Section 3) and transforming its content, which is a string, into a Prolog term (call to the user-defined predicate select). The select predicate calls the unpack predicate provided by the TuJadeLibrary in order to transform the string that represents the content of the message into a term, and calls the user-defined handle predicate on the message performative, the obtained term, and the message sender.

We only illustrate the case that the message received had content bought(Goods). The buyer agent knows the price of Goods (fact price(S, Goods, P)) and it knows the amount of quantity of Goods it bought (fact buys(goods(Goods), quantity(Q))). Since he succeeded in buying Goods, it must update both the possessed amount of Goods and the remaining money (calls to standard Prolog predicates retract, is

and `assert`). Similar definitions of the predicate `handle` are provided for the other kinds of message that the buyer agent can receive.

The ordinary JADE agent, `seller`, is characterised by a Java code partly shown in Figure 7. The `Seller` class extends the `JADE Agent` class. The behavior of the seller agent is a cyclic behavior (`class SellBehaviour extends CyclicBehaviour`) which continuously checks for a message (`msg = myAgent.receive()`) and, if the message is present, handles it (`if (msg != null) handleMsgs(msg)`).

Once all the agents have been specified using `tuProlog` or `JADE`, they can be loaded into `JADE` and the execution of the obtained prototype can start. `JADE` offers the possibility to follow the communication between the agents by means of the “sniffer” agent whose output is shown in Figure 8.

PER IVANA: cambia la figura dello sniffer in modo che ci siano i 5 agenti, e poi cancella questa parte di testo!!!

The state of the agents’ mailboxes can be inspected thanks to the introspector agent. Figure 9 shows the state of the mailbox of `buyer2`. This screenshot has been taken at the beginning of the simulation; all the `INFORM` messages showed there are answers to price requests previously issued by `buyer2` to the seller agents.

Details on the messages exchanged can also be inspected. Figure 10 shows the request for the price of kiwi sent by `buyer2` to `seller1`. Figure 11 shows the answer to this request.

The execution and monitoring of the prototype carried out by exploiting the tools provided by `JADE`, allow the developer to see that the agents work well with respect to their intended behavior. The output of the sniffer agent allows to verify that the messages are exchanged in the expected order (for example, that all the buyers ask for the price of fruit first, and start buying fruit afterwards), while the detailed views of the messages allow to verify that the content of replies is consistent with the content of requests. Without the integration of `tuProlog` into `JADE`, verifying the correct working of communicating agents implemented in `Prolog` could only be done by hand: the developer had to put breakpoints in its code or he/she had to write messages on the standard output or on a file in order to follow what was going on during the prototype execution. `CaseLP` offers graphical debugging tools more sophisticated than this “by-hand” inspection. Nevertheless, the adoption of the instruments already provided by a standard, `FIPA`-compliant and open-source platform, represents an improvement with respect to the use of proprietary instruments offered by `CaseLP`.

5 Conclusions and future work

Scrivo poche righe dopo pranzo e te le mando verso le 14. Tu inseriscile QUI

References

1. E. Astesiano, M. Martelli, V. Mascardi, and G. Reggio. From Requirement Specification to Prototype Execution: a Combination of a Multiview Use-Case Driven Method and Agent-Oriented Techniques. In J. Debenham and K. Zhang, editors, *Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering (SEKE'03)*, pages 578–585. The Knowledge System Institute, 2003.

2. E. Astesiano and G. Reggio. Knowledge Structuring and Representation in Requirement Specification. In *Proceedings of SEKE 2002*. ACM Press, 2002.
3. E. Astesiano and G. Reggio. Tight Structuring for Precise UML-based Requirement Specifications: Complete Version. Technical Report DISI-TR-03-06, DISI, Università di Genova, Italy, 2003.
4. M. Baldoni, C. Baroglio, L. Giordano, A. Martelli, and V. Patti. Reasoning about communicating agents in the semantic web. In F. Bry, N. Henze, and J. Maluszynski, editors, *Proceedings of the 1st International Workshop on Principle and Practice of Semantic Web Reasoning (PPSWR 2003)*, pages 84–98. Springer-Verlag, 2003. Volume 2901 of LNCS.
5. M. Baldoni, C. Baroglio, I. Gungui, A. Martelli, M. Martelli, V. Mascardi, V. Patti, and C. Schifanella. Reasoning about communicating agents inside DCaseLP. Submitted, 2004.
6. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about interaction protocols for web service composition. In M. Bravetti and G. Zavattaro, editors, *Proceedings of the 1st International Workshop on Web Services and Formal Methods (WS-FM 2004)*. Elsevier Science Direct, 2004. Electronic Notes in Theoretical Computer Science.
7. M. Baldoni, L. Giordano, A. Martelli, and V. Patti. Programming rational agents in a modal action logic. *Annals of Mathematics and Artificial Intelligence*, Special issue on Logic-Based Agent Implementation. To appear.
8. F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. JADE – a white paper. Available at <http://jade.cselt.it/papers/WhitePaperJADEEXP.pdf>, 2003.
9. Jess home page. <http://herzberg.ca.sandia.gov/jess/>.
10. M. Martelli and V. Mascardi. From UML diagrams to Jess rules: Integrating OO and rule-based languages to specify, implement and execute agents. In F. Buccafurri, editor, *Proceedings of the 8th APPIA-GULP-PRODE Joint Conference on Declarative Programming (AGP'03)*, pages 275–286, 2003.
11. M. Martelli, V. Mascardi, and F. Zini. Caselp: a prototyping environment for heterogeneous multi-agent systems. Available at <http://www.disi.unige.it/person/MascardiV/Download/aamas-journal-MMZ04.%ps.gz>.
12. SICStus Prolog home page. <http://www.sics.se/isl/sicstuswww/site/index.html>.
13. FIPA Specifications. <http://www.fipa.org>.
14. TuProlog home page. <http://lia.deis.unibo.it/research/tuprolog/>.

```

main :-
    handle_msgs,
    ask_prices,
    buy_goods.

goods_posessed( oranges, 0 ) :- true.
goods_posessed( apples, 0 ) :- true.
goods_posessed( kiwi, 0 ) :- true.

buys( goods( oranges ), quantity( 2 ) ) :- true.
buys( goods( apples ), quantity( 3 ) ) :- true.
buys( goods( kiwi ), quantity( 12 ) ) :- true.

money( 200 ) :- true.

sellers_addresses( [ "seller1@gruppoai:1099/JADE",
                    "seller2@gruppoai:1099/JADE",
                    "seller@gruppoai:1099/JADE" ] ) :- true.

.....

handle_msgs :-
    receive( Performative, Message, Sender ),
    select( Performative, Message, Sender ).

select( Performative, Message, Sender ) :-
    bound( Performative ),
    bound( Message ),
    address_name( Sender, Name ),
    unpack( Message, TermMsg ),
    handle( Performative, TermMsg, Sender ).

select( _, _, _ ) :- true.

handle( "INFORM",
        bought( Goods ),
        Sender ) :-
    bound( Goods ),
    address_name( Sender, S ),
    price( S, Goods, P ),
    retract( money( M ) ),
    retract( goods_posessed( Goods, X ) ),
    buys( goods( Goods ), quantity( Q ) ),
    N is X + Q,
    P <= N,
    NM is M - P,
    assert( money( NM ) ),
    assert( goods_posessed( Goods, N ) ).

```

Fig. 6. A piece of the tuProlog theory associated to buyer1.

```

package tuPinJADE;

import jade.core.Agent;
import jade.core.AID;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;

public class Seller extends Agent
{
    private int orangesAmount = 5;
    private int applesAmount = 5;
    private int kiwiAmount = 10;
    private int orangesPrice = 105;
    private int applesPrice = 80;
    private int kiwiPrice = 100;

    protected void setup()
    {
        SellBehaviour p = new SellBehaviour(this);
        addBehaviour(p);
    }
}

class SellBehaviour extends CyclicBehaviour
{
    private static boolean done = false;

    public SellBehaviour(Agent a)
    {
        super(a);
    }

    public void action()
    {
        ACLMessage msg;
        while (!done)
        {
            msg = myAgent.receive();
            if (msg != null) handleMsgs(msg);
        }
    }
}

.....

```

Fig. 7. A piece of the Java code that defines seller.

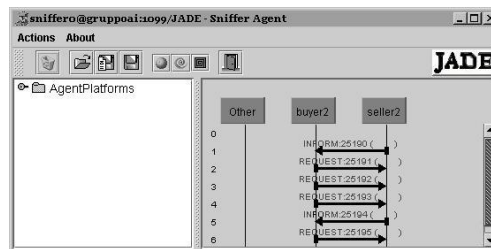


Fig. 8. Output of the JADE sniffer agent.

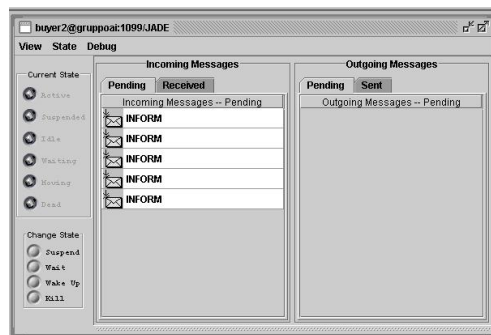


Fig. 9. JADE window showing the communication among agents.

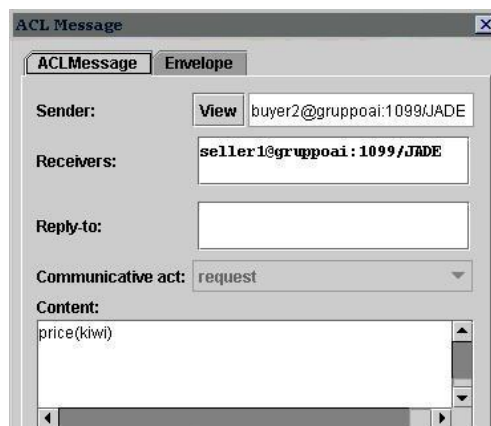


Fig. 10. Price request from buyer2 to seller1.

ACLMessage Envelope

Sender: [View](#) seller1@gruppoai:1099/JADE

Receivers: buyer2@gruppoai:1099/JADE

Reply-to:

Communicative act: inform

Content:
price(kiwi,120)

Fig. 11. Price answer from seller1 to buyer2.