International Journal of Software Engineering and Knowledge Engineering
Vol. 12, No. 6 (2002) 605–618

SUPPORTING AGENT-ORIENTED MODELLING WITH UML

FEDERICO BERGENTI and AGOSTINO POGGI
Dipartimento di Ingegneria dell'Informazione,
Università degli Studi di Parma,
Parco Area delle Scienze, 181/A, I-43100 Parma, Italy

Software engineering relies on the possibility of describing a system at different levels of abstraction. Agent-oriented software engineering introduces a new level of abstraction, that we called agent level, to allow the architect modelling a system in terms of interacting agents. This level of abstraction is not supported by an accepted set of tools and notations yet, even if a number of proposals are available. This paper introduces: (i) An UML-based notation capable of modelling a system at the agent level and (ii) A development framework, called ParADE, exploiting such a notation. The notation we propose is formalized in terms of a UML profile and it supports the realisation of artefacts modelling two basic concepts of the agent level, i.e., the architecture of the multi-agent system and the ontology followed by agents. The choice of formalising our notation in terms of a UML profile allows using it with any off-the-shelf CASE tool. The ParADE framework takes advantage of this choice by providing a code generator capable of producing skeletons of FIPA-compliant agents from XMI files of agent-oriented models. The developer is requested to complete the generated skeletons exploiting the services that ParADE and the underlying agent platform provide.

Keywords: Agent modeling; agent-oriented notation; FIPA; UML.

1. Introduction

The ever-increasing importance of the Web is promoting the development of agent technology and it is driving the introduction of *agent-oriented software engineering* [8, 16, 25, 28]. Agent technology is advocated as an ideal means to exploit the possibilities of the Web [7] and agent-oriented software engineering has been accepted in the software engineering community with the AOSE (Agent-Oriented Software Engineering) workshop at ICSE and with the creation of the OMG Agent Special Interest Group. Surprisingly, this wave of interest has not yet produced accepted tools and notations capable of supporting the new concepts that agent introduces in software engineering. In this paper we present some tools that we propose to overcome this problem. In particular, we show:

- An UML-based notation that can be used to create agent-oriented artefacts;
- A development framework, called ParADE, capable of exploiting such artefacts.

Our notation covers two basic concepts of agent-oriented software engineering: the architecture of the multi-agent system and the ontology followed by agents. We support such concepts by introducing two new UML models in the framework of a new UML profile [14]. This approach is well-founded because it relies on the extension mechanism that UML provides and it allows using any CASE tool to employ our notation. Moreover, the availability of XMI [15], the standard XML-based file format for exchanging UML models between CASE tools, allows realising CASE-agnostic tools supporting our notation. The ParADE framework exploits this file format to produce skeletons for agents relying on a goal-oriented agent architecture. A description of ParADE code generators can be found in [4].

In order to define precisely the agent-oriented concepts that we support in our tools, the following section briefly characterises software agents. Then, in Sec. 3 we build a UML profile capable of capturing the concepts that we used in our characterisation. Section 4 describes the ParADE framework and shows how this implements such concepts. Finally, Sec. 5 discusses the presented work and presents some conclusions.

2. Software Agents

The agent community has not yet adopted an accepted definition for the word *agent*. Many informal definitions are available, but none of them focuses on all the elements that we want to support in our notation. This is the reason why we need to provide our characterization for agents starting from Wooldridge's work [27]:

Definition. An *agent* is a software system that is (i) situated in some environment, (ii) capable of autonomous actions in order to meet its objectives, and (iii) capable of communicating with other agents.

Agents are software systems that work in an environment formed by non-agentised entities. Agents take the essential resources they need to work from their environment. This characteristic emphasizes that agents are intended to work together with non-agentised entities to bring about their objectives.

The second characteristic that we allow for agents is autonomy: agents are explicitly associated with goals and they are capable of taking autonomous decisions to bring about them without any external coercion. This characteristic is very important to deal with open and dynamic environments like the Web because it allows agents reacting to unpredicted and/or unpredictable situations.

The third characteristic that we associate with agents is the so-called social ability, i.e., the ability that agents have to cooperate. This characteristic relies on communication. Various communication models are available in the literature, and we adopt the model that FIPA (the Foundation for Intelligent Physical Agents) [11] designed to promote interoperability [13] between agents realised by different developers. This communication model is a variation of the approach developed by the KSE (Knowledge Sharing Effort) [10, 19] initiative and it derives from the

speech-act theory [21]. The basic idea of the FIPA communication model is that an agent can explicitly send messages to other agents and no implicit communication is taken into account. Such messages belong to the FIPA ACL (Agent Communication Language) [11] and each message is structured in two parts: a communication part and a content part. The communication part carries content-independent information such as the identifier of the sender and the receiver. The content part bares the meaning of the message and it is structured in terms of one performative and a domain content. The performative is a domain-independent verb that specifies the meaning that the sender of the message wanted to associate with the domain content in terms of the intention that guided the sender in sending the message. FIPA specifies a set of performatives that cover most of the intentions an agent may have to send a message as it allows expressing variants of queries, requests and information exchanges. The domain content of the message is a sentence expressed in some language and its semantics is in the scope of some ontology. FIPA does not prescribe any language for expressing domain contents and when an agent receives a message it comes to know the language and the ontology used in the domain content exploiting the communication part of the message.

An important characteristic of the FIPA communication model is that it is not limited to isolated messages, but it allows grouping messages into interaction protocols [11]. Interaction protocols are domain-independent message flows that can be instantiated in concrete conversations between agents. They abstract a common path of performatives from the set of possible conversations and they are designed to ease the realisation of agents capable of complex conversations. FIPA provides a set of general-purpose interaction protocols and it also provides guidelines for the realisation of application-specific interaction protocols.

3. UML Profile for Agent-Oriented Software

Software engineering relies on the possibility of modelling a system at different levels of abstraction. Agent-oriented software engineering introduces a new level of abstraction, that we called *agent level* [3], to model a system in terms of interacting agents. At this level, an agent is an atomic entity that communicates with other agents and interacts with its environment to implement the functionality of the system. The development of complex agent-based systems requires the extension of the available development processes to take the peculiarities of the agent level into account. A number of processes are already available [8, 28], but at the moment none of them is supported by a diagrammatic notation. We address this problem introducing a UML profile that can be used to model two important concepts of the agent level: the ontology followed by agents and the architecture of the multi-agent system. The notation that we propose differs significantly from other proposals [1, 6, 9] found in the literature because:

- It is well founded as it is coherent with the UML extension mechanism;
- It can be exploited using available tools.

Artefacts using such a notation can be produced with any off-the-shelf UML CASE tool and we developed CASE-independent software tools relying on XMI.

Table 1 shows the stereotypes and the associated elements in the UML metamodel that we introduce to formalise our notation. We introduce two new models, called ontology model and architecture model, that are concretely realised as constrained class diagrams called ontology diagrams and architecture diagrams. An ontology model describes the ontology followed by the agents in the system. It is described in terms of a top-level package called ontology system. This package can contain ontology packages and/or classes and relationships. An ontology package is a package containing classes and relationships.

Architecture models are used to model the architecture of the multi-agent system and they are structured like ontology models. Any architecture model is described in terms of a top-level package called architecture system. This package can contain architecture packages and/or classes and relationships. An architecture package recursively contains other architecture packages and/or classes and relationships.

The following subsections describe the constraints that we impose on classes and relationships to complete our profile.

Table 1. Stereotypes supporting ontology and architecture diagrams.

Metamodel class	Stereotype name
Model	ontology model
Model	architecture model
Package	ontology system
Package	architecture system
Package	ontology package
Package	architecture package
Class	entity
Class	agent
Association	communicate
Association	predicate

3.1. *Ontology diagrams*

Modelling a multi-agent system at the agent level requires defining a model of the environment in which agents execute. Agents exploit this model to reason about the environment and to talk about it. The problem of describing the environment to agents is traditionally solved by providing them with an ontology that models the environment in terms of entities and relationships between such entities [9, 12]. Ontology diagrams are class diagrams whose semantics is similar to the one employed in conceptual diagrams. An ontology diagram allows describing the entities belong-

ing to an ontology in terms of *entity classes*. These classes are characterised only in terms of public attributes. These attributes are used to model the structure of the entities comprised in the ontology, just like they are used in conceptual diagrams. Moreover, ontology diagrams allow the architect defining the relationships between the entities belonging to an ontology by exploiting relationships between entity classes. These are mapped into UML by exploiting public relationships between entity classes.

The agent level of abstraction does not deal with implementation details because these are normally tackled at a lower level of abstraction, that we called *object level* [3], where agents are seen as object-oriented systems. This implies that entity classes are not allowed to contain private and protected methods and attributes. Similarly, class diagrams allow associating a set of public methods with a class of objects to specify the messages objects may exchange to implement the functionality of the system. This is not allowed at the agent level because the actors communicating to implement the system are the agents and not the entity belonging to the ontology.

Ontology diagrams contain the main features needed to support the communication between agents, i.e., they contain a subset of the vocabulary of domain contents, because we treat the relationships between entity classes as predicates defined over such classes. This approach allows the architect to exploit an ontology diagram as a description of the entities and the predicates that agents may use to create domain contents. To this extent, we introduce the stereotype *predicate* for association between entity classes.

Figure 1 shows an ontology diagram we modelled for the realisation of an agent-based system managing a restaurant service. This service will be probably included in the services provided by the Agentcities initiative [26]. It comprises various entity classes and predicates and it is rich enough for modelling complex domain contents. The stereotype predicate is not shown because all relationships in the diagram are predicates.

3.2. *Architecture diagrams*

All proposed agent-oriented development processes emphasise the importance of modelling a multi-agent system in terms of a society of related agents. This introduces the concept of architeture of a multi-agent system. Such architecture is composed of a set of *agent classes* and each class represents a class of agents with a precise set of responsibilities [17]. A complete model of an agent class requires describing the set of features used to characterise an agent belonging to that class. Such features include:

- The accepted content messages, taking into account the ontology model;
- The supported interaction protocols;
- The network of acquaintance an agent can have.

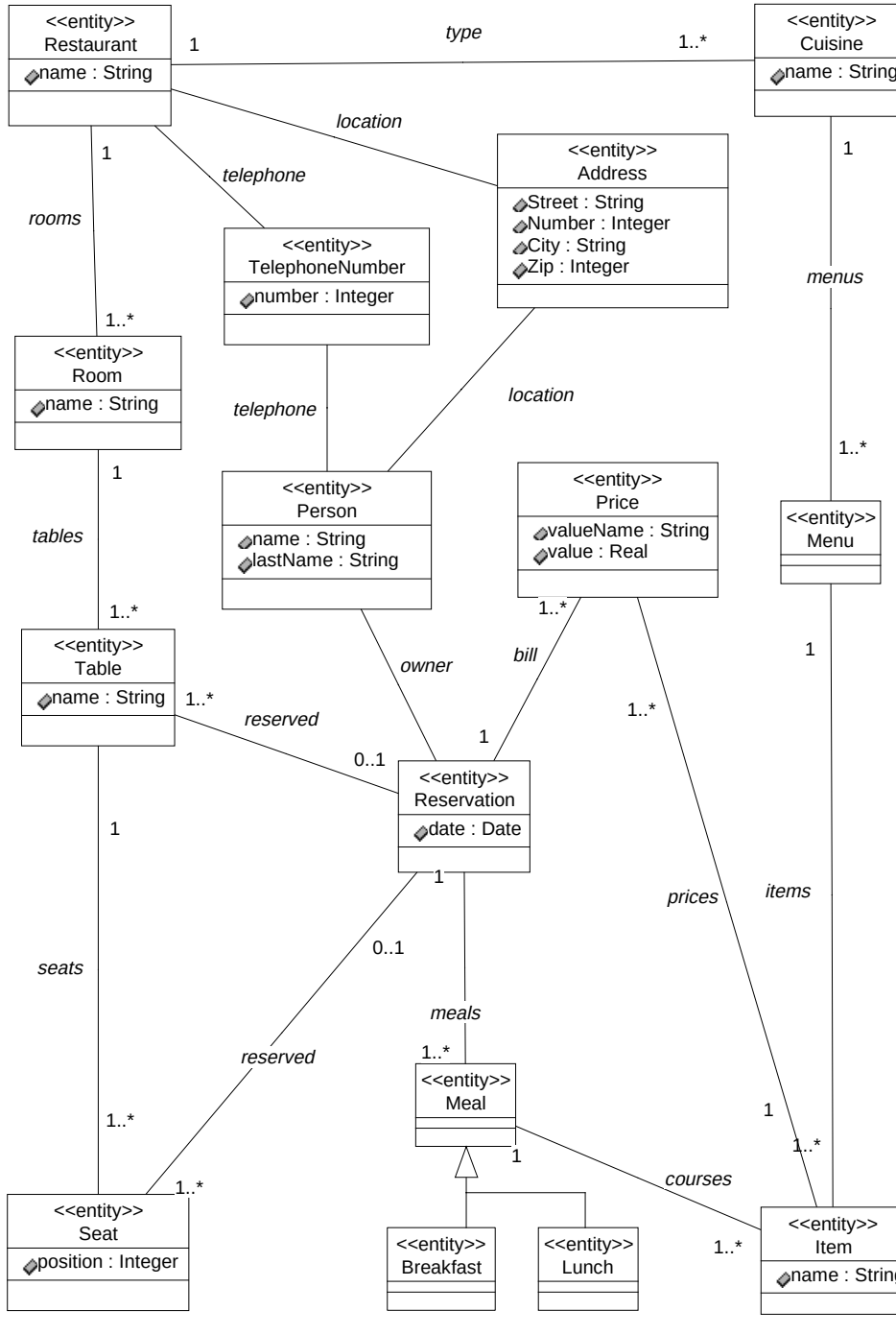

Fig. 1. Ontology diagram for an agent-based restaurant service.

Architecture diagrams allow modelling such characteristics with class diagrams where classes can contain only public methods. These methods correspond to the actions that an agent belonging to that class can be requested to perform and they must be declared *void* because in the FIPA communication model messages are asynchronous, i.e., they are not explicitly associated with a reply. Moreover, the parameters passed to these methods must belong to entity classes defined in the ontology model. The list of actions associated with an agent class does not include the performatives of the agent communication language because these are almost embedded in the chosen agent model.

The actions that the architect associates with agent classes complete the elements provided by the ontology model to create valid domain contents. An agent can understand all messages composed using the entities and predicates found in the ontology model and can be requested to perform the actions specified in its class.

This approach restricts agents understanding only messages based on first-order logic. While this is a limitation of the considered logic framework, it does not limit strongly the set of messages that agents may wish to exchange in a real-world system. Therefore, the development of a more comprehensive logic framework is considered for a future work.

The problem of modelling application-specific interaction protocols has been investigated by Odell *et al.* [1, 6], but we believe that allowing agents using application-specific interaction protocols may cause problems from the interoperability point of view. Even if we provide agents with a run-time description of an interaction protocol, it is extremely difficult to implement an agent capable of taking such a description and learn how to use it without any explicit help from the developer. Therefore, agents using application-specific protocols may not be able to run in open and dynamic systems where third-party agents join and leave dynamically. Moreover, the semantics of the paths of an interaction protocol must be coherent with the semantics of the employed performatives. This coherence is very difficult to achieve and only well-studied and accepted interaction protocols can guarantee this property. FIPA provides a set of generic interaction protocols that can be composed to create complex interactions and we take the assumption that agents use only these interaction protocols. Nevertheless, FIPA states that no interaction protocol is mandatory and we need a way to express which interaction protocol each agent supports. We do this by informally annotating agent classes with structured comments.

Architecture diagrams use public relationships to express relationships between agents. It is quite common to avoid distributing responsibilities to agents using relationships because this approach may be problematic in systems where agents can leave and join dynamically. Therefore, we use relationships between agent classes only to model the privileged communication relationships and we introduce the stereotype *communicate*.

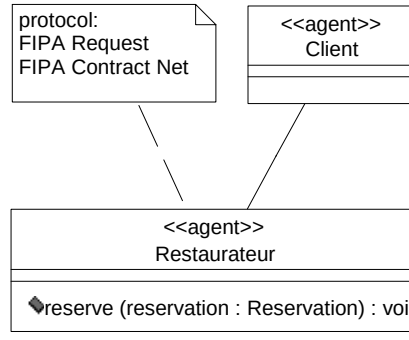

Fig. 2. Architecture diagram for an agent-based restaurant service.

Figure 2 shows the small architecture diagram that we build for our restaurant service. It contains only two agent classes and only restaurateur agents can perform non-communicative actions. The stereotype communicate is not explicit in the diagram because normally all relationships in architecture models are of this type.

4. The ParADE Framework

The need of an agent-oriented support for programmers was felt long before the introduction of agent-oriented software engineering. The first attempt to provide this support were agent-oriented programming languages [22]. Nowadays, the approach of agent-oriented languages is no longer adopted and the greater majority of agents are developed in Java exploiting one of the available agent platforms [2, 5, 20, 24, 23]. Besides, agent platforms are only runtime environment designed to support the agent lifecycle and to provide a communication mechanism. They do not provide any support for autonomy and the only support for interoperability that they grant are the services for producing and parsing correct messages in some agent communication language. Therefore, programmers cannot exploit easily the concepts introduced at the agent level.

We address this problem implementing an agent-development framework called *ParADE* (*Parma Development Environment*). This framework is intended to provide tools for easily implementing autonomous and interoperable agents over the Jade [2] FIPA-compliant platform. The goal driving the work on ParADE is providing the developer with a support capable of exploiting the UML notation that we have just introduced in the scope of a hybrid agent architecture that tries to balance between goal-orientation and reactiveness to achieve a good trade-off between autonomy, interoperability and performances. This approach was chosen because goal-orientation is a fundamental key in supporting autonomy while reactive agents are easier to design and to implement.

ParADE is composed of a set of development tools supporting the developer at the agent level and at the object level. At the agent level, the developer can produce UML models as described in this paper and ParADE can generate code

for the skeletons of agents. This code relies on the ParADE development library and on the services provided by the underlying agent platform. It is worth noting that ParADE does not integrate any CASE tool because the code generator works with XMI files that any off-the-shelf CASE tool should be able to produce.

The code that ParADE generates relies on an goal-oriented agent architecture that integrates also reactive behaviours. A number of goal-oriented architectures are available in the literature and a review of the more important ones can be found in [27]. Nevertheless, such architectures are not meant to exploit FIPA specifications and therefore they do not integrate the semantics of FIPA ACL and they do not take FIPA generic interaction protocols into account. The idea behind our approach is to exploit goal-orientation to assembly plans composed of actions and FIPA generic interaction protocols. Plans are built and scheduled autonomously, i.e., starting from the current goals of the agent, but during the execution of an interaction protocol the agent is reactive and it simply responds to incoming messages. If, during the execution of a plan, an action or a protocol fails, then the whole plan is dropped and the agent needs to reconsider how to satisfy its goals [18]. Three basic advantages that derive from this approach are:

- Our agents are autonomous because they are driven only by their goals and therefore they can cope well with dynamic environments;
- Our agents are interoperable because we use only FIPA generic interaction protocols and we assemble them into complex interaction exploiting their formal semantics;
- We promote efficiency because agents exploit the semantics of FIPA ACL without the need for reasoning on how to build an interaction protocol in terms of isolated communicative acts.

Our architecture relies on the possibility of generating plans to achieve the goals of the agent. To this extent, the planning engine is provided with a description of the roles that the agents in the system play. This description is generated from the architecture model of the system and contains:

- The actions that the agent can perform;
- The generic interaction protocols that the agent supports.

Similarly to other models found in the literature, actions and protocols are characterized in terms of a feasibility precondition and a post-condition. For the case of actions, the feasibility precondition states what must be true for that action to be feasible. This allows agents to decide whether they can perform an action or not, but it does not impose on the agent to perform that action if it does not intend to do it. The post-condition of an action asserts what is certainly true after the execution of that action. For the case of a protocol, the feasibility precondition is a generalization of the feasibility precondition for the actions. In particular, the feasibility precondition of a protocol states what must be true for an agent to initiate that protocol. The definition of post-conditions for protocols requires noting that

all FIPA generic interaction protocols are characterized by one success state and one failure state. Sometimes such states are graphically repeated in the diagrams used in FIPA specifications, but this is only a drawing convention and it does not mean that the protocol can end in more than two different states. Success and failure states represent success or failure of the protocol from the point of view of the initiator.

The availability of feasibility preconditions and post-conditions for actions and for the generic interaction protocols allows defining the planning engine. We use protocols as plan templates and we use their post-conditions to decide when we should employ a particular protocol. We say that a protocol is a plan template because it must be instantiated providing application-specific domain contents.

The planning engine is in charge of building a sequence of protocols and actions capable of satisfying a chosen goal starting from the current state of the environment and the current state of the agent. This is a classic planning problem and it has been studied intensively in the literature. Therefore, we can access to a huge set of techniques that can be adopted to solve this problem. In the current implementation of ParADE we provide an implementation of the simplest of such techniques and a future work is intended to evaluate the adoptability of more sophisticated planning algorithms. Figure 3 shows a coarse-grained pseudo-code of the planning engine currently implemented in ParADE. First, the engine verifies if the goal is currently asserted in the knowledge base and if so it stops the planning process. Otherwise, it analyses the goal to see whether it contains references to other agents or not. If the goal does not contain any reference to another agent, then the planner looks for an action to execute in the space of the actions the agent can perform. In the case that the goal refers to other agents, the planning engine searches for a protocol that may satisfy the goal. Choosing an action or a protocol simply means unifying the current goal with the post-condition of such action or protocol. The problem of instantiating the protocol is solved by this unification, as it allows associating a value with the initial proposition of the protocol. The remaining propositions are instantiated during the execution of the protocol because the agent behaves reactively in these situations. The algorithm shown in figure 3 recursively builds plans until a first plan leading to the goal is found. This algorithm uses the priorities associated with protocols to sort the set of protocols unifying the current goal.

Even if the presented planning technique is very simple, it may fit many application scenarios. Nevertheless, the current implementation of ParADE allows integrating any planning engine as long as the appropriate Java interface is implemented.

The proposed agent architecture is split in two parallel threads. A main thread runs the main loop of the architecture, while a second thread, called messaging thread, waits for messages on the agent's mailbox and changes the knowledge base taking into account the semantics of FIPA ACL. These threads do not interact directly because the information produced by the messaging thread is stored in the knowledge base and the main thread observes only the changes of the knowledge

```
boolean planner(goal, knowledgebase, plan)
  if knowledgebase asserts goal then
    return true
  end if

  if goal contains only Me then
    space = my actions
  else
    space = my interaction protocols
  end if

  forall action in space whose post-
         condition unifies goal
    queue action to plan
    assert goal in knowledgebase

    success = planner(action precondition,
                      knowledgebase, plan)

    if success then
      /* stop at first plan */
      return true
    else
      remove action from plan
      deny goal in knowledgebase
    end if
  end forall

  /* if here no action can be found */
  return false
end plan
```

Fig. 3. Pseudo-code of ParADE planning engine.

base. When a message is taken from the agent's mailbox, the messaging thread asserts that the sender of the message intended to achieve the rational effect of the communicative act and it also asserts the feasibility precondition of this act. Such assertions rely on two assumptions: the feasibility precondition does not take time into account and agents are rational, i.e., they want to achieve the rational effect of the communicative acts they perform. The first assumption is coherent with the logic framework that ParADE provides to applications, while the second is a fundamental assumption of FIPA compliancy.

First, the agent waits for new goals or for new actions to perform. New goals and new actions may result from the execution of other actions or from the arrival of messages. A message within the scope of a protocol stimulates the execution of an action to continue the protocol, while messages outside the scope of protocols cause changes in the knowledge base and may stimulate goals. FIPA does not constraint the behavior of an agent receiving a message outside the scope of a protocol and therefore the application developer can choose what to do without breaking

interoperability. In the current implementation, ParADE reacts to this kind of messages putting the rational effect of the incoming message as a new goal. This is a cooperative behaviour that the developer can change simply extending the Java class implementing the agent template.

Once new goals or new actions become available, the main loop tests if new goals are available. If this is the case, the planning engine is invoked and possibly a new plan is generated. If a new plan can be generated, then the first action of the plan is scheduled. In the case that no new goals are available, then an action is ready to be executed. The main loop performs this action and receives the next action of the plan in return. If such an action exists, then it is scheduled, otherwise the plan is finished in the success or failure state. If the plan ended in the success state, then the corresponding goal can be asserted, otherwise the goal remains pending and the agent may decide to achieve it later. This is the only point where agents reconsider their intentions [18]. The decision on whether a goal is unreachable or not is completely application-specific and therefore the main loop calls a Java method that the developer can re-implement to test the achievability of goals.

5. Conclusions

Agent-oriented software engineering introduces the agent level of abstraction to allow the architect describing a system in terms of interacting agents. At this level of abstraction, an agent is considered as an atomic entity that communicates with other agents to implement the functionality of the system. Besides, the architect may not exploit the benefits offered by this new level of abstraction for the lack of accepted notations and tools. This paper shows a new UML profile that can be used to model a system at the agent level. This profile introduces two new models for modelling the ontology that agents follow and the architecture of the multi-agent system. We support this notation with a development framework called ParADE that can be used to implement FIPA-compliant agents. These agents rely on a goal-oriented agent architecture designed as a compromise between autonomy, interoperability and performances. ParADE can be used both at the agent level and at the object level. At the agent level, it provides a code generator capable of producing Java skeletons for agents. The generated skeletons must be completed with application-specific code at the object level exploiting a development library and the services offered by the underlying agent platform. For a deeper description of ParADE internal architecture can be found in [4].

Acknowledgements

This work is partially supported by CSELT and by the European Commission through the contracts IST-1999-12217, *CoMMA - Corporate Memory Management through Agents* and IST-1999-10211, *LEAP — Lightweight Extensible Agent Platform.*

References

1. B. Bauer, J. P. Müller, and J. Odell, "An extension of UML by protocols for multiagent interaction", in *Proc. ICMAS 2000*, Boston, 2000.
2. F. Bellifemine, A. Poggi, and G. Rimassa, "Developing multi-agent systems with JADE", in *Proc. ATAL 2000*, Boston, 2000.
3. F. Bergenti and A. Poggi, "Exploiting UML in the design of multi-agent systems", in *Proc. ESAW Workshop at ECAI 2000*, Berlin, 2000.
4. F. Bergenti and A. Poggi, "A development toolkit to realize autonomous and interoperable agents", in *Proc. Agents 2001*, Toronto, 2001.
5. P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas, "JACK intelligent agents — components for intelligent agents in Java", in *AgentLink Newsletter*, 1999.
6. FIPA Technical Committee C, "Extending UML for the specification of agent interaction protocols", available at `http://www.fipa.org`.
7. L. Chiariglione, "Helping agent technologies get across to the market place", available at `http://www.fipa.org`.
8. MESSAGE Consortium, "Deliverable 1: Initial methodology", 2000, deliverable of the EURESCOM Project P907-GI.
9. S. Cranefield and M. Pruvis, "UML as an ontology modelling language", in *Proc. Workshop on Intelligent Information Integration*, 1999.
10. T. Finin and Y. Labrou, "KQML as an agent communication language", in *Software Agents*, MIT Press, 1997.
11. FIPA, "FIPA 99 specification part 2: Agent communication language", available at `http://www.fipa.org`.
12. M. R. Genesereth and R. E. Fikes, *Knowledge Interchange Format — Version 3: Reference Manual*, Stanford University, Stanford, 1992.
13. M. R. Genesereth, N. Singh, and M. Syed, "A distributed and anonymous knowledge sharing approach to software interoperation", *Int. J. Cooperative Information Systems* **4(4)** (1995) 339–367.
14. Object Management Group, "OMG unified modeling language specification", available at `http://www.omg.org`.
15. Object Management Group, "OMG XML metadata interchange (XMI) specification", available at `http://www.omg.org`.
16. C. A. Iglesias, M. Garijo, and J. C. A. González, "Survey of agent-oriented methodologies", in *Proc. ATAL 1998*, 1998.
17. E. A. Kendall, "Agent roles and role models: New abstractions for multiagent systems analysis and design", in *Proc. Int. Workshop on Intelligent Agents in Information and Process Management*, 1998.
18. S. Parsons, O. Pettersson, A. Saffiotti, and M. Wooldridge, "Intention reconsideration in theory and practice", in *Proc. ECAI 2000*, Berlin, 2000.
19. R. S. Patil, R. E. Fikes, P. F. Patel-Scheneider, D. McKay, T. Finin, T. Gruber, and R. Neches, "The DARPA knowledge sharing effort: Progress report", in *Proc. Third Conference on Principles of Knowledge Representation and Reasoning*, Cambridge, 1992.
20. S. Poslad, P. Burckle, and R. Hadingham, "The FIPA-OS agent platform: Open source for open standards", 1999, available at `http://fipaos.sourceforge.net`.
21. M. D. Sadek, "Dialogue acts are rational plans", in *Proc. ESCA/ETRW Workshop on the Structure of Multimodal Dialogue*, Maratea, 1991.
22. Y. Shoham, "An overview of agent-oriented programming", in *Software Agents*, MIT Press, 1997.
23. M. Straßer, J. Baumann, and F. Hohl, "MOLE — a Java based mobile agent system",

in *Special Issues in Object Oriented Programming*, Verlag, 1997.
24. "Reticular Systems. Agentbuilder — an integrated toolkit for constructing intelligent software agents", 1999, available at `http://www.agentbuilder.com`.
25. J. Treur, "Methodologies and software engeering for agent systems", 1999, available at `http://www.cs.vu.nl/`.
26. S. Willmot and B. Burg, "Agentcities testbed coordination workplan", available at `http://www.agentcities.org`.
27. M. Wooldridge, "Intelligent agents," in G. Weiss (ed.), *Multiagent Systems*, MIT Press, 1999.
28. M. Wooldridge, N. R. Jennings, and D. Kinny, "The gaia methodology for agent-oriented analysis and design", *Journal of Autonomous Agents and Multi-Agent Systems* **3(3)** (2000) 285–312.