

A Development Toolkit to Realize Autonomous and Inter-operable Agents

Federico Bergenti and Agostino Poggi

Dipartimento di Ingegneria dell'Informazione

Università degli Studi di Parma

Parco Area delle Scienze 181/A, 43100 Parma, Italy

Tel. +390521905708, Fax +390521905723

{Bergenti, Poggi}@CE.UniPR.IT

ABSTRACT

Autonomy and inter-operability are two characteristics of software agents that are advocating agent technology as an ideal candidate to support next generation of software systems. This paper presents a Java development toolkit supporting the realization of autonomous and inter-operable agents. This toolkit provides the developer with a goal-oriented agent architecture for FIPA-compliant agents. Goal-orientation supports autonomy because the developer is no longer requested to describe what the agent should do in reaction to events. To this extent, our development toolkit provides a planning engine capable of building plans to achieve the agent's goals autonomously. Goal-orientation is also a key aspect of inter-operability because it is the basis of the semantics of FIPA ACL. Our development toolkit can be used at two levels of abstraction. The higher level, that we call the agent level, allows describing the agent in terms of its natural characteristics such as goals, beliefs and social organization. A code generator producing Java skeletons from UML diagrams supports this level. The developer can choose any UML CASE tool to model her agents because this code generator works with files in a standard format. The generated skeletons must be completed with application-specific code at the lower level of abstraction, that we call object level. At this level, agents are seen as Java programs and the developer is provided with a development library to integrate her code within the generated skeletons. This two-level approach allows describing agents in their natural terms at the agent level, while supporting the integration of application-specific and legacy code at the object level. Moreover, the generated code can be customized at the object level to integrate application-specific optimizations.

Keywords

Agent tools, agent architectures, agent-based software engineering, standards for agents, AUML, FIPA.

1. INTRODUCTION

The increasing importance of the Web in everyday life is pushing the need of software capable of coping with open and dynamic environments. More than other technologies, agents seem to have the necessary characteristics to support the development of open and flexible software systems. In particular, two characteristics of agents seem more important in this perspective: *autonomy* and *inter-operability*. Autonomy is necessary when dealing with dynamic environments to allow agents adapting to unpredictable situations. We cannot delegate to the developer the prediction of all possible courses of events in the Web. Inter-operability is a necessary condition for the implementation of open systems [9][10][21]. Every second new information and new services become available on the Web and we cannot wait for developers evolving their products to take advantage of these new opportunities.

The development tools currently available to agent developers fail in supporting both autonomy and inter-operability. Nowadays, the development of agents and multi-agent systems is based on two kinds of tools: agent platforms and BDI-like tools. Agent platforms, such as JADE [1] and FIPA-OS [22], provide only a transport layer and some basic services, but they do not provide any support for autonomy. Moreover, they lack support for inter-operability because they do not take into account the semantics of their agent communication language both in the case of FIPA ACL [8] and KQML [6]. We call this kind of inter-operability *syntactic inter-operability* because it relies only on parsing and generating correct messages. In order to cope with the open nature of the Web we need *semantic inter-operability* between agents and this can be achieved only taking into account the semantics of the adopted agent communication language. The available BDI-like tools, such as dMARS [13], JACK [4], AGENTBUILDER [23], JAM [11] and ZEUS [16], support only syntactic inter-operability because they do not exploit their reasoning engines to integrate the semantics of the adopted agent communication languages. Moreover, they are not always goal-oriented because the reactive approach is often preferred as it is considered easier to use in large-scale projects. As a matter of facts, the developer is requested to provide autonomy and inter-operability to her agents with only a limited support from the development tools.

This paper describes an agent development toolkit called *PARADE* (*Parma Development Environment*) that we implemented to provide a support for autonomy and inter-

operability over existing FIPA-compliant platforms. Even if PARADE could have been implemented over any existing agent platform, its current implementation works on top of JADE [1] and takes advantage of its services. The goal driving the work on PARADE is providing the agent developer with a hybrid agent architecture capable of promoting inter-operability and supporting autonomy exploiting the semantics of FIPA ACL [8][24]. Such an architecture is basically goal-orientated but it also integrates reactive behaviors. This approach is chosen because goal-orientation is a fundamental key in supporting both autonomy and inter-operability, and we think that reactive agents are easier to design and to implement. Goal-oriented agents are inherently autonomous because they act to achieve their goals without requiring the developer to foresee all possible flows of events. Moreover, the semantics of FIPA ACL is specified in terms of a goal-oriented agent model and therefore its integration with an agent platform is straightforward exploiting a goal-oriented agent architecture. It is worth noting that the integration of the semantics of FIPA ACL within a development tool is a step forward the current generation of FIPA-compliant development tools. In fact, current tools allows parsing and generating correct messages, but they do not take into account why an agent should send a message and what the receiver agent should do with that message.

PARADE is composed of a set of development tools supporting the developer at two levels of abstraction. The higher level, that we call the *agent level*, allows describing agents in terms of their natural characteristics such as beliefs, goals and social organization. At this level of abstraction, the developer produces UML models describing the multi-agent system, the agents and the ontology that agents follow. PARADE provides a code generator capable of compiling such models into Java code implementing skeletons of the agents in the system. This code relies on PARADE development library and on the services provided by the agent platform. It is worth noting that PARADE does not provide any CASE tool because the code generator works with XMI [18] files that any off-the-shelf CASE tool should be able to produce. The code generated from agent-level models is only a skeleton and the developer is requested to complete it at the *object level* integrating application-specific behaviors. Such behaviors are implemented exploiting the PARADE development library to access PARADE components such as the knowledge base or the planning engine. This two-level approach has the advantage of supporting the production of code from UML models at the agent level without taking implementation details into account. Moreover, it allows integrating legacy code and supporting application-specific optimizations at the object level.

The rest of this paper describes PARADE and its underlying agent architecture. In particular, section 2 analyses PARADE agent architecture showing its basic concepts and its main execution loop. Section 3 presents the tools the developer can exploit to build agents. In particular, it introduces the UML notation that we use to model agents at the agent level and the development tools provided at the object-level. Finally, section 4 discusses the presented tools and its underlying ideas.

2. THE AGENT ARCHITECTURE

PARADE provides a hybrid agent architecture that exploits both goal-orientation and reactive-ness to support autonomy and inter-operability without disregarding efficiency and user friendliness.

In the previous section we emphasized the need of goal-orientation to support autonomy and inter-operability. Obviously, goal-oriented agents are also autonomous because their behavior should not be explicitly programmed but it is deduced from their beliefs and goals. Moreover, we believe that the only feasible way to manage the semantics of FIPA ACL is to exploit its declarative nature using a goal-oriented agent architecture. Therefore, autonomy and inter-operability suggest employing a purely goal-oriented architecture, but the well-known efficiency problems of this approach cannot be ignored.

A number of goal-oriented architectures are available in the literature and a review of the more important of them 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 presented agent architecture is a hybrid architecture based on the ideas underlying the classic logic-based architectures and that integrates the semantics of FIPA ACL and FIPA generic interaction protocols. In particular, we exploit goal-orientation to assembly plans composed of actions and generic interaction protocols. Plans are built and scheduled autonomously, but during the execution of a protocol the agent is reactive because it simply reacts to the incoming messages. This allows the developer driving the agent reactively in know interaction paths, i.e., during the execution of protocols. 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. Two basic advantages derive from this architecture: we can exploit autonomy and inter-operability and we promote efficiency because the agent no longer needs to reason on how to build a protocol in terms of isolated communicative acts.

The proposed agent architecture relies on the possibility of describing an agent in terms of a mental state, a set of possible actions and a set of supported FIPA generic interaction protocols. The mental state is composed of a set of beliefs, a set of persistent goals and a set of transient goals. Persistent goals represent the intentions of the agent, i.e., the goals it is committed to achieve and that it will try to achieve until it comes to believe that they are reached or that they are unreachable. Transient goals are goals that the agent constructs as intermediate steps to achieve a persistent goal. Basically, they are sub-goals build by the planning engine to achieve a persistent goal. Beliefs, transient goals and persistent goals are sets of propositions describing the mental state of the agent. Such propositions can contain the following elements:

- *done* operator that associates an agent to an action that it performed;
- agent identifiers, exploiting FIPA agent identifier [8];
- *exist* and *forall* operators defined over the beliefs of the agent;
- variables, that can be free in goals and must be bound in beliefs;
- logic connectives;
- entities and predicates defined in the ontology.

Such elements constraint the logic framework provided to the application developer. This may limit the systems that can be implemented using PARADE, but we believe that such limitations may not affect the majority of applications. Moreover, PARADE provides customization points in the generated code where the developer can explicitly provide support for more sophisticated

reasoning, such as enforcing temporal constraints or explicitly representing beliefs and goals to the agent.

The planning engine uses actions and generic interaction protocols to support its reasoning process. This engine is the core of the proposed agent architecture and it is responsible for producing plans of actions and protocols starting from the current agent's beliefs and goals. To this extent, the planning engine is provided with a description of the roles that agents can play in the system. This description contains:

- the actions that an agent playing such role can be requested to perform;
- the generic interaction protocols that an agent playing such role supports.

Similarly to the models found in BDI-like architectures, actions and protocols are characterized in terms of a *feasibility precondition* and a *post-condition*. Such conditions are expressed as propositions containing the elements we allowed for goals and therefore they can contain free variables. For the case of actions, the feasibility precondition states what must be true for that action to be feasible. This allows agents deciding whether they can perform an action or not, but it does not impose the agents to perform that action if they do 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 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. In fact, success and failure states represent success or failure of the protocol from the point of view of the initiator. For example, the FIPA contract net protocol [8], a variant of the contract-net protocol [25], can have two outcomes:

- either one contractor informs the manager that it performed the requested action, and therefore the protocol ends in the success state;
- or no contractor performs the action, for one of the admissible reasons, and the protocol ends in the failure state.

Recognizing that FIPA generic interaction protocols have always one success state and one failure state allows defining the post-condition of an interaction protocol as a proposition that is certainly true for the initiator of the protocol when the protocol finishes in the success state. In the case of the FIPA contract net protocol, an agent decides to initiate this protocol if it wants an action to be performed by some other agent. Therefore, the feasibility precondition states that the initiator believes that somewhere in the multi-agent system it can find an agent capable of performing such an action. The post-condition states that finishing the protocol in the success state guarantees that one of the contractors performed the action. The described conditions do not express completely the meaning of the FIPA contract net protocol. For example, the post-condition does not state that the action was performed under the best available conditions for the manager. We do not introduce such a level of detail in our model of the FIPA contract net because of two reasons: it might cause the agent spending too much time in planning and it would

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

  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 = planer(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
```

Figure 1. Pseudo-code of the planning engine provided with the current implementation of PARADE.

require a more sophisticated logic model capable of describing conditions like “best price”. Even if this limitation seems reasonable, especially from the point of view of efficiency, it may lead two protocols to have exactly the same post-condition. For example, the FIPA request protocol and the FIPA contract net protocol are interchangeable for the planning engine. This is the reason why we allow the developer prioritizing the protocols. Moreover, the current implementation of PARADE provides hooks in the planning engine that can be used to provide application-specific code intended to discriminate between equivalent protocols.

The availability of feasibility preconditions and post-conditions for the actions that an agent can perform and for the generic interaction protocols it may use to communicate 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-

```

void main(knowledgebase)
  forever
    wait for goals or
      for an action to perform

    if goals are available then
      goal    = choose a goal in goals
      success = planner(goal,
                        knowledgebase, plan)

      if success then
        schedule the first action of
          the plan
      end if
    else
      nextaction = perform action

      if nextaction exist then
        schedule nextaction
      else
        if action failed the plan then
          drop the plan

          if the goal is unreachable then
            drop the goal
          end if
        else
          assert the goal
        end if
      end if
    end if
  end forever
end main

```

Figure 2. The main loop of PARADE agents.

specific content messages. 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 world and the current mental 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 1 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 1 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. Transient goals are not explicit because they are the first parameter of the `plan` method and therefore they are created and destroyed during the planning process.

Even if the presented planning technique is very simple, it may fit many application scenarios, especially concerning information agents. In fact, many information agents simply use well-known protocols to extract information from information sources. Therefore they do not need to build complex plans, but they need to search quickly in their library of plans. Nevertheless, the current implementation of PARADE allows providing a different planning engine simply wrapping it within a subclass of the Java class `PlanningEngine`.

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 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.

Figure 2 shows the main loop of PARADE agents. 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 constraints 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 inter-operability. This circumstance is coherent with autonomy, but it leads to interesting drawbacks as discussed in section 4. 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 behavior 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 [20]. 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. If the developer needs to perform reasoning on the knowledge base to decide on the achievability of a goal, then she can use the generic reasoning engine described in the following section.

3. THE DEVELOPMENT SUPPORT

PARADE supports the developer at two levels of abstraction: the agent level and the object level. The agent level considers agents as atomic entities that communicate to implement the functionality of the system. This is the reason why describing a system at the agent level means modeling the architecture of the multi-agent system and the elements agents use to communicate. We defined in [2] a UML notation to model a multi-agent system at this level of abstraction and PARADE provides a tool to generate code for agents from models employing such a notation. At the object level each agent is seen as an object-oriented system and PARADE provides an object-oriented development library to implement agents at this level.

3.1 Working at the Agent Level

The first step in the development of a PARADE system consists in modeling the essential characteristics of such a system at the agent level. Modeling a multi-agent system at the agent level requires modeling the architecture of the multi-agent system and the interactions between agents. Such elements can be modeled using UML exploiting what we call *architecture diagrams* and *ontology diagrams*.

Architecture diagrams are UML class diagrams used to model the roles [14] that agents play in the system. Each role is represented by a class that we call *agent class*. We introduce the stereotype *agent* to tag the classes in architecture diagrams to ease the implementation of tools managing such diagrams. Associations between agent classes describe possible associations between agents playing different roles. Such associations are basically used to express acquaintance, as it is common to promote flexibility in multi-agent systems avoiding the use of associations to spread responsibilities across agents.

An agent class can be used to associate a set of actions and a set of generic interaction protocols with a role. Actions are represented as public methods of the class. Such methods must be declared *void* because no concept of return value is defined for actions. Moreover, the parameters of such actions must belong to classes of entities defined in ontology diagrams. The list of actions associated with an agent class does not include the performatives of the agent communication language because PARADE agents are FIPA-compliant and therefore they employ FIPA ACL.

Object protocols, the object-oriented counterpart of interaction protocols, are not included in UML because they are not considered a valuable concept in object orientation. This is the reason why we propose to annotate agent classes with the list of supported interaction protocols exploiting comments. This allows modeling only known protocols and the developer is implicitly requested to use only generic interaction protocols. The motivation of this choice is simply to preserve inter-operability. In fact, introducing application-specific interaction protocols may lead to the following problems. Even if we provide agents with a run-time description of a protocol, it is extremely difficult to implement an agent capable of taking such a description and learning how to use the protocol without any explicit help from the developer. Therefore, agents using application-specific protocols may not be able to run in open systems where third-party agents join and leave the system dynamically. Moreover, in order to support inter-operability, the semantics of the paths of a protocol must be coherent with the semantics of the employed performatives. This coherence is very difficult to achieve and only well-studied and accepted protocols can guarantee this properties. These are the reasons why we believe that complex interactions should be obtained composing FIPA generic interaction protocols and therefore we do not consider the integration of AUML interaction-protocol diagrams [7][17] in PARADE.

Comments are used in architecture diagrams also to model feasibility preconditions and post-conditions of actions. Such conditions are expressed exploiting the predicates defined in the ontology and the *believe* operator. Such a representation is very simple and its expressive power is limited, therefore we are currently investigating the possibility of using OCL [19] to model such conditions.

PARADE uses architecture diagrams to generate the code of agent skeletons. Each agent class is associated with one agent skeleton. Many agents can be instantiated from a single skeleton, but the current implementation of PARADE imposes the restriction that the class of an agent cannot change in time. This limits agents to play always the same role within the system and a future work will be devoted to investigate a possible solution to such a problem preserving efficiency. Figure 1(a) shows the architecture diagram of a simple example inspired by e-commerce. It shows two classes of agents: personal assistants and CD shop agents. Agents belonging to such classes support the FIPA contract net protocol the FIPA request protocol. Moreover, CD shop agents can be requested to perform an action called *sell*. Such an action requires the description of the personal assistant buying the CD, the description of the CD it wants to buy and its conditions of payment. CD shop agents declare the semantics of the *sell* action stating that if the desired CD is still available, then after the execution of this action the personal assistant will have the CD.

Ontology diagrams modify the notation introduced by CraneField and Purvis [5] to make it more useful for generating code for agents. An ontology diagram is a class diagram where classes model classes of entities in defined in the ontology. As for architecture diagrams, we tag the classes in ontology diagrams, i.e., *entity classes*, with the stereotype *entity*. Entities are structured using public attributes. Ontology diagrams allow defining relations between classes and we use such relations to model the predicates provided by the ontology. Figure 1 (b) shows the ontology of the CD shop example. This diagram comprises three classes of entities: CDs, prices and payment

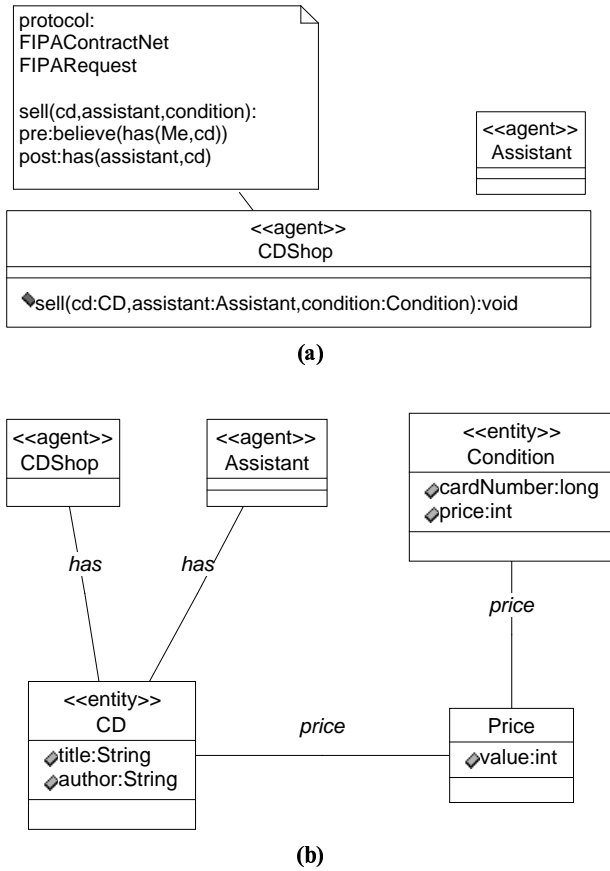


Figure 2. (a) Architecture diagram and (b) ontology diagram of the CD shop example.

conditions. CDs are characterized by a title and an author, prices are characterized by an integer value while payment conditions comprise a credit card number and a price. The diagram shows also four predicates: two predicates *has* and two predicates *price* defined over different entities.

PARADE code generator exploits ontology diagrams to build a set of classes representing the entities and the predicates in the ontology. Such classes are intended to support the developer in managing the concepts of the ontology at the object level. Moreover, they are used by generic components in the PARADE development library to integrate the concepts of the ontology.

Ontology diagrams and architecture diagrams provide a diagrammatic notation to represent the concepts introduced in the agent architecture. Architecture diagrams allow modeling the actions and the interaction protocols that a class of agents supports. Actions are annotated with their feasibility precondition and their post-conditions. Ontology diagrams complete the elements that agents may use to form propositions introducing entities and predicates.

3.2 Working at the Object Level

PARADE can be used at the agent level to model the multi-agent system exploiting the diagrams described in the previous section. PARADE code generator can produce Java code from such models and the developer is requested to complete the generated skeletons providing application-specific code. This two-level approach allows integrating legacy code and providing optimizations.

Moreover, it allows the developer choosing whether to express application-specific behaviors in terms of the supported logic framework or, more explicitly, in terms of Java methods.

The code produced by the code generator relies on the services provided by the platform and on the PARADE development library. The core class of the PARADE development library is class `Agent`. This class is the agent template that implements the agent architecture described in the previous section and the developer subclass it to complete the skeleton of the agent with application-specific code. Moreover, this class can be extended to customize or to add new functionality to the agent architecture. In particular, it can be extended to provide new implementations of the key concepts of the agent architecture such as the planning engine or the knowledge base. Extending the `PlanningEngine` class allows providing agents with a more sophisticated planning algorithm than the simple algorithm described in the previous section. Subclasses of the `KnowledgeBase` class can be provided to add deduction rules to agents. Every manipulation of the knowledge base is associated with a hook method that the developer can use to provide application-specific code to be executed just before, or just after, this manipulation. If the manipulation is a query, the developer can provide deductions to manage the result of the query. If the manipulation is an insertion or a deletion from the knowledge base, the developer can provide deductions to add or to remove new predicates from the knowledge base. This approach is chosen because we wanted to limit the unnecessary deductions that a PARADE agent may perform. Therefore, we delegated to the application developer the choice of what to deduce and when to deduce it. In order to support the developer in performing deductions, the PARADE development library provides a simple PROLOG-like rule system, called *generic reasoning engine*, that the developer can use to build facts and rules within the scope of the knowledge base.

PARADE code generator takes as input XMI [19] files that any an off-the-shelf CASE tool should be able to produce. The generated code is composed of application-specific subclasses of the general-purpose classes provided by the PARADE development library. In particular, subclasses of the classes `Entity`, `Predicate` and `Agent` are generated exploiting the information contained in architecture and ontology diagrams. The subclasses of the `Entity` and `Predicate` classes provide a concrete implementation for the entities and the predicates defined in the ontology. The subclasses of `Entity` allow treating the entities of the ontology as Java objects and, in particular, they allow accessing the attributes of the entities as Java fields. This simplifies the use of the entities defined in the ontology in the application-specific code. Similarly, subclasses of `Predicate` allow treating predicates defined in the ontology as Java objects. While this representation allows treating all predicates uniformly, it is not very convenient when using predicates in application-specific code. This is the reason why PARADE code generator synthesizes a subclass of the `OntologyHelper` class to provide a more direct access to predicates. This class is combined with the `Agent` class to provide an application-specific agent template where programmers can treat predicates as Java methods. As an example, the code generated from the ontology diagram shown in figure 2 allows the developer writing the following piece of Java code to make a CD shop agent informing a personal assistant of the price for a CD:

```

CD    cd          = new CD(title, author);
Price newPrice = new Price(valueOfThePrice);

send(personalAssistant,
    inform(price(cd, newPrice)));

```

Another important component of the PARADE development library is the ontology-driven parser [3] for the SL0 [8] language. This is a general-purpose component that the application developer does not need to customize. It takes as input a run-time description of the entities and the predicates defined in the ontology and parses incoming SL0 messages taking such concepts into account. The PARADE code generator synthesizes the run-time description of the ontology needed by the parser. The output of the SL0 parser is a Java object describing the input message in terms of the classes built by the code generator. This allows the programmer dealing only with the elements of the ontology rather than with the abstract syntax tree of the incoming message. Providing the developer only with the elements of the ontology does not only simplify the application code, but it also allows writing code that does not depend on the content language used for communications. Input messages are transparently parsed by the agent template and only the result of the parsing is passed to the application. Similarly, output messages are generated from classes created by the code generator and the developer is not required to encode the messages in a particular content language. The current implementation of PARADE integrates only the parser for the SL0 language, but the developer can provide new parsers implementing the Parser class.

4. DISCUSSION AND CONCLUSIONS

This work presents an implemented agent development toolkit called PARADE intended to show the benefits of using hybrid architectures in supporting autonomy and inter-operability. In particular, PARADE agent architecture supports autonomous agents capable of exploiting FIPA compliancy at the semantic level rather than only at the syntactic level. Moreover, PARADE takes efficiency into account using FIPA generic interaction protocols to implement reactive behaviors. It is worth noting that the semantics of FIPA ACL seems to allow using isolated communicative acts as planning primitives rather than protocols. Unfortunately, such an approach is not only less efficient than using protocols but it also prohibits the use of agents in open environments because it requires some hypotheses on the characteristics of the agents in the system. The semantics of FIPA ACL allows agents choosing the communicative acts to perform on the basis of feasibility preconditions and rational effects. Such conditions describe what must be true for an act to be feasible and the result that a rational agent performing that act wishes to obtain. This description is not sufficient for planning purposes because there is no guarantee that an agent receiving a message will take this message into account. For example, there is no guarantee that requesting another agent to perform an action will result in such an agent performing the action or just informing of the refusal. In fact, the receiver of a message has no obligations with respect of the rational effect intended by the sender of this message. Therefore, an agent cannot use rational effects to infer the effects of an action and it cannot rely on rational effects to plan a sequence of actions. This is not really a limitation of the semantics of FIPA ACL because it is coherent with the characteristic autonomy of agents. Nevertheless, this prohibits using isolated communicative acts to build planning agents. This

limitation can be overcome exploiting the semantics that FIPA provides for interaction protocols. In particular, FIPA specifications state that if an agent supports a FIPA generic interaction protocol, then all conversations it runs with another agent using such a protocol must be finished according to this protocol [8]. This implies, for example, that if an agent requests to another agent to perform an action within the FIPA request protocol, then the receiving agent must perform the action and inform the sender or it must explicitly refuse to perform the action. This constraint was introduced to allow developers dealing with interaction protocols easily, but it is fundamental to support goal-oriented agents because it allows defining post-conditions for protocols.

The basic difference between the PARADE approach and all FIPA compliant tools available in the literature is the exploitation of the semantics of FIPA ACL to support autonomy. This approach overcomes a basic problem of many applications developed using such tools: they need to limit agent expressive power. In particular, common limitations are: performatives cannot be nested in a message and some new performative must be introduced to support application-specific concepts. Nested performatives are not allowed because, normally, a Java method is associated with each performative or each step of a protocol. Nevertheless, the meaning of a message containing nested performatives is not calling two methods. Basically, this problem comes from treating messages as imperative sentences rather than declarative assertions. The need of introducing application-specific performatives is a consequence of the problem described above. The impossibility of nesting performatives implies the impossibility of creating macro-performatives and therefore the developer may not find a performative with the desired meaning within FIPA ones. A possible solution to such a problem could be associating agents with actions that have the same post-condition of the desired performative. This solution moves the problem from the domain of agent communication language to the domain of the content language and therefore it implies including concepts such as beliefs and intentions in the ontology. Even if this might be a working solution, it may be pushed to its extreme consequences of using just the request performative delegating the meaning of messages completely to the content. This approach is quite similar to the communication scheme of distributed object-oriented programming and therefore it lacks many of the benefits of available agent communication languages.

Another important advantage that we see in the presented approach is that it seems to help bridging the gap between specification and implementation. In fact, the description of agents we introduced in section 2 is quite similar to the one employed in the methodologies supporting the specification of multi-agent systems [12], such as GAIA [28], MESSAGE [15] and UAML [26]. Such methodologies are not yet supported by any tool and therefore the developer has no means to generate code from specification or design artifacts. PARADE provides an implemented tool along with a general-purpose agent architecture and this may help using it in real-world applications.

PARADE has currently a number of limitations deriving both from its underlying theoretic model and from its current implementation. Nevertheless, PARADE provides concrete support to the developer and the possibility of using UML diagrams to generate code should promote its use also in the industry. Moreover, PARADE is a step forward the available FIPA-compliant tools because it integrates the semantics of FIPA

ACL to support semantic inter-operability and autonomy without disregarding efficiency.

5. ACKNOWLEDGMENTS

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*.

6. REFERENCES

- [1] Bellifemine, F., Poggi, A. and Rimassa, G. “*Developing multi-agent systems with JADE*”, in Proceedings of ATAL 2000, Boston, 2000.
- [2] Bergenti, F. and Poggi, A. “*Exploiting UML in the Design of Multi-Agent Systems*”, in Proceedings of ECAI workshop ESAW’00, Berlin, 2000.
- [3] Bergenti, F. and Poggi, A. “*A Development Environment for the Realization of Open and Scalable Multi-Agent Systems*”, in Proceedings of MAAMAW’99, Valencia, 1999.
- [4] Busetta, P., Rönquist, R., Hodgson, A. and Lucas, A. “*JACK Intelligent Agents - Components for Intelligent Agents in Java*”, in AgentLink News Letter, 1999.
- [5] Cranefield, S. and Purvis, M. “*UML as an Ontology Modelling Language*”, in Proceedings of the Workshop on Intelligent Information Integration, 1999.
- [6] Finin, T., Labrou, Y. and Mayfield, J. “*KQML as an Agent Communication Language*”, in Bradshaw J. M. (Ed.) *Software Agents*, MIT Press, 1997.
- [7] FIPA Technical Committee C “*Extending UML for the Specification of Agent Interaction Protocols*”, response to the OMG Analysis and Design Task Force UML RTF 2.0 Request for Information, 1999.
- [8] FIPA “*FIPA ’99 Specification Part 2: Agent Communication Language*”, available at <http://www.fipa.org>.
- [9] Genesereth, M. R. and Fikes, R. E. “*Knowledge Interchange Format - Version 3 - Reference Manual*”, technical report Logic-92-1, Stanford University, 1992.
- [10] Genesereth, M. R., Singh, N. and Syed, M. “*A Distributed and Anonymous Knowledge Sharing Approach to Software Interoperation*”, International Journal of Cooperative Information Systems 4(4):339-367, 1995.
- [11] Huber, M. J. “*JAM: A BDI-theoretic Mobile Agent Architecture*”, in Proceedings of Agents 1999, Seattle, 1999.
- [12] Iglesias, C. A., Garijo, M. and González, J. C. A. “*Survey of Agent-Oriented Methodologies*”, in Proceedings of ATAL’98, 1998.
- [13] d’Inverno, M., Kinny, D., Luck, M. and Wooldridge, M. “*A Formal Specification of dMARS*”. In Singh, M. P., Rao, A. S. and Wooldridge, M. (Eds.) *Intelligent Agents IV*, Springer-Verlag LNAI, vol. 1365, 1998.
- [14] Kendall, E. A. “*Agent Roles and Role Models: New Abstractions for Multiagent System Analysis and Design*”, International Workshop on Intelligent Agents in Information and Process Management, 1998.
- [15] MESSAGE Consortium, “*Deliverable 1: Initial Methodology*”, deliverable of the EURESCOM Project P907-GI, 2000.
- [16] Nwana, H. S., Ndumu, D. T. and Lee, L.C. “*ZEUS: An advanced Toolkit for Engineering Distributed Multi-Agent Systems*”, in Proceedings of PAAM’98, London, 1998.
- [17] Odell, J. and Bock, C. “*Suggested UML Extensions for Agents*”, response to the OMG Analysis and Design Task Force UML RTF 2.0 Request for Information, 1999.
- [18] OMG, “*XML Metadata Interchange – Version 1.1*”, available at <http://www.omg.org>.
- [19] OMG, “*Unified Modeling Language Specification – Version 1.3*”, available at <http://www.omg.org>
- [20] Parsons, S., Pettersson, O., Saffiotti, A. and Wooldridge, M. “*Intention Reconsideration in Theory and Practice*”, in Proceedings of ECAI 2000. Berlin, 2000.
- [21] Patel-Schneider, P. F. and Swartout, B. “*Description-Logic Knowledge Representation System Specification*”, DARPA KSE technical report, 1993.
- [22] Poslad, S., Buckle, P., Haddingham, R. “*The FIPA-OS Agent Platform: Open Source for Open Standards*”, available at <http://fipa-os.sourceforge.net>.
- [23] Reticular Systems “*AgentBuilder - An Integrated Toolkit for Constructing Intelligence Software Agents*”, available at <http://www.agentbuilder.com>.
- [24] Sadek, M. D. “*Dialogue acts are rational plans*”, in Proceedings of the ESCA/ETRW Workshop on the Structure of Multimodal Dialogue, Maratea, 1991.
- [25] Smith, R. “*The Contract Net Protocol: High-Level Communications and Control in a Distributed Problems Solver*” in IEEE Transactions on Computers, vol. 29, no. 12, 1980.
- [26] Treur, J. “*Methodologies and Software Engineering for Agent Systems*”, available at <http://www.cs.vu.nl/~treur>.
- [27] Wooldridge, M. “*Intelligent Agents*”, in Weiss G. (Ed.) *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, MIT Press, 1999.
- [28] Wooldridge, M., Jennings, N. R. and Kinny, D. “*The Gaia Methodology for Agent-Oriented Analysis and Design*”. In Journal of Autonomous Agents and Multi-Agent Systems, 3(3):285-312, 2000.