A Plug-in Architecture Providing Dynamic Negotiation Capabilities for Mobile Agents

M.T. Tu, F. Griffel, M. Merz, and W. Lamersdorf

Distributed Systems Group, Computer Science Department,
University of Hamburg
Vogt–Kölln–Str. 30, D–22527 Hamburg
{tu, griffel, merz, lamersd}@informatik.uni-hamburg.de
http://vsys-www.informatik.uni-hamburg.de

Abstract. The diversity of research and development work on agent technology has led to a strong distinction between *mobile* and *intelligent* agents. This paper presents an architecture aiming at providing a step towards the *integration* of these two aspects, concretely by providing an approach of dynamically embedding *negotiation capabilities* into mobile agents. In particular, the requirements for enabling automated negotiations including negotiation protocols and strategies, a plug-in component architecture for realizing such requirements on mobile agents, and the design of negotiation support building blocks as components of this architecture are presented.

1 Introduction

In recent years, the development of agent technology has drawn particularly great attention of people working in very different fields of computer science such as distributed systems, artificial intelligence, system management and electronic commerce. This at a first glance quite surprising fact is essentially based on the appealing general supposition that agents are *autonomous* entities which can perform tasks assigned to them independently, i.e. completely without user intervention. Thus, even if there is no universally accepted definition of software agents, autonomy is commonly considered one of the most important features of agenthood. Due to this growing interest, the agent programming paradigm has made considerable progress and is going to be well established, especially through agent systems based on Java and WWW technologies such as [7, 12, 17, 9] etc..

However, a consequence of the diversity of research and development work on agent technology is that a strong distinction between "mobile" and "intelligent agents" has emerged which can also be regarded as a distinction between *location autonomy* and *decision autonomy.* This simple de-facto classification of agents at present seems unfortunately inaccurate because the features of mobility and intelligence are obviously by no means mutually exclusive or even only contrary. And although the integration of these two qualities is certainly desired in order to face the challenging requirements of *realistic* agent application fields such as mobile/asynchronous computing, information retrieval, electronic commerce etc., there have been up to now very few concrete approaches in this direction. This paper presents an architecture aiming at providing a step towards such an integration, concretely by providing an approach of dynamically embedding negotiation capabilities into mobile agents. Before going into this approach, some general requirements arising from the integration problem should be mentioned: One of the main difficulties a practical approach has to deal with is that the incorporation of "intelligent" capabilities into mobile software agents can become very expensive because reasoning mechanisms, for instance, are usually much more complex than a few "go" and "select" commands coded in simple agents roaming the network nowadays, i.e. the agents can become literally too "fat" and consequently, their mobility is reduced. This problem is even more severe if the concrete purpose or task of the agent is not known ahead (at compilation time), in which case either many agents for different tasks or very general-purpose agents, which are likely even bigger in size, have to be built. In order to cope with this problem, some important requirements have to be imposed on a corresponding system design:

- Role-specific functionality: A mobile agent should not be loaded with every kind of available functionality or intelligent capability at the same time (as it is usually the case with complex AI systems or human beings), but should rather carry with him only the functionality required to fill out the actual *role(s)* assigned to him at a given time, for instance "seller" or "notary" in the context of electronic commerce.
- On-the-fly loading: Moreover, the functionality of an agent should be able to be loaded "on demand", i.e. at (or short before) the moment it is really needed.
- Flexible configuration: The agent's functionality should also be flexibly and dynamically configurable so that it can be reused in many similar, but differently constrained situations without having to replace its corresponding implementation.

In order to satisfy these requirements, a flexible *plug-in* component architecture has been designed and is being implemented in the DYNAMICS project at University of Hamburg. It is used as the framework to dynamically embed the negotiation capabilities for mobile agents presented in this paper, the rest of which is organized as follows: In section 2, the requirements for enabling automatic negotiations, especially those concerning negotiation protocols and strategies, are identified. Section 3 then outlines the plug-in architecture of DYNAMICS. Next, the building blocks to support automated negotiations are described in section 4. Implementation issues are discussed in section 5 and finally, the paper is concluded by section 6.

2 Requirements for Enabling Automated Negotiations

One of the most obvious motivations for developing agent technology is to provide agents which are able to perform commercial transactions on behalf of the people launching them (see, e.g., [4]). Using mobile agents in electronic commerce is attractive for several reasons: disburdening people from routine transactions, handling (gathering, selecting) great amounts of information in a given time, supporting mobile device users by asynchronous communication etc. (see [1]). Such agents would be even more useful if they could autonomously negotiate a deal in case the default assumptions of the participants about the desired transaction do not exactly match, much analogous to the way people carry out business negotiations. Indeed, with negotiating agents, the possible benefit of negotiations, i.e. finding the best possible options for a transaction, would likely be employed much more frequently because of the low cost the agents raise in comparison to the cost of human negotiators. However, in order to enable software agents to carry out automatic negotiations, the process of negotiation must first be formally specified by a respective *protocol* and, for each agent, a *strategy* of producing the proper negotiation actions needs to be implemented. In the following, we will discuss the requirements for formal negotiation protocols and strategies.

2.1 Negotiation Protocols

To make software agents interact in a meaningful way with the purpose of reaching an agreement, a set of rules must be defined which constrain the possible interactions between the participating agents. The formal specification of the rules applying to the interactions during a negotiation is usually called a negotiation *protocol*. Even in most conventional negotiations, a set of more or less explicit rules has to be followed to maintain a meaningful course of the negotiation process. More explicit rules apply, for example, in case of an auction or an advertised bidding, less explicit ones in case of a car purchase. Concerning automated negotiations, the protocol has to be precise and extensive enough to cope with all situations that may occur during a negotiation, even with those that are not expected in human interactions. Furthermore, it has been shown that the design of negotiation protocols may have quite sophisticated influence on the strategic behavior of participants (see, e.g., [15]) and is also therefore worth to be investigated thoroughly.

In order to develop a general approach to implement negotiation protocols, the first question to be raised is which aspects of a negotiation can be regulated or constrained by a protocol. Indeed, these aspects are manifold and can refer to:

- *Issue*: A negotiation can have one or more issues, each of which is associated with a set of fixed or negotiable attributes.
- *Participants*: Constraints applying to participants concern following subaspects
  - Roles: In many negotiation types, participants have fixed roles which determine their type of relationship to the issues, e.g., customer, and therefore which actions they can take.

  – Quantity: For each role, it is to specify how many instances need to (at minimum) or can (at maximum) be involved.
  – Admission and exclusion: Conditions of when to admit and to exclude participants need to be specified, e.g., whether it is possible for additional parties to enter an on-going negotiation.
- *Validation*: The validity of the actions taken by the participants is to be checked, particularly with regard to the syntactical and semantical correctness of offers submitted.
- *Proceeding*: Constraints applying to the proceeding of a negotiation include following sub-aspects
  – Round definition and number: A negotiation process can be specified in terms of rounds, i.e. periodical phases of exchanging offers and counter-offers, in which case it must be precisely specified what constitutes a round as well as the min. and max. number of rounds to be performed.
  – Voting method: In order to determine the agreement of the involved parties, some voting method has to be applied which is to be specified by the protocol.
  – Timeout: In general, every action of the participants needs to be assigned a timeout period and timeout handling measures are to be specified.
  – Truncation condition: The protocol has to specify when the negotiation process is terminated, in case of success as well as in case of failure.
- *Bindingness*: In case of success, the result of a negotiation is not always binding to all participants, e.g., in case of an auction. On the other hand, the negotiation result could also apply to participants who have not voted for it, as in case of a shareholders' meeting, for example. Therefore, the bindingness of negotiation results needs to be specified precisely.

There exist some formal mathematical treatments of negotiation protocols which can serve as a nice theoretical basis to understand some formal aspects of negotiation protocols (see, e.g., [11]). However, such models usually do not consider several aspects listed above such as admission and exclusion of participants, round definition and number, timeout and bindingness. In general, this kind of *static* modeling alone does not seem appropriate to capture the dynamic aspects of a negotiation process. Therefore, we have chosen a Petri-net based workflow management approach to handle negotiation protocols in the DYNAMICS project described below.

2.2 Negotiation Strategies

Although negotiation protocols have the purpose of restricting the possible courses of negotiation, they must obviously leave alternatives for participating parties to choose from. In choosing between or proposing protocol-compliant alternatives, each participant follows its own negotiation *strategy* which is normally not disclosed to other parties. Thus, in order to enable automated negotiations using agents, it is neccessary to equip each agent with a formalized strategy to compute actions and offers corresponding to the role it takes in the negotiation.

Formalization Criteria In order to design a general framework to implement such negotiation strategies, it is necessary to examine the criteria that are relevant for their development first. (In 4.3 we will see how such criteria are used to specify the interfaces of a strategy component.)

- *Utility function*: Intuitively, the goal of a negotiation strategy is achieving "good" results, which is achieved mainly by producing good offers. Therefore, *utility functions* are required to evaluate offers according to many possible criteria, for example price or quality of goods, time to negotiate etc.. Usually, a combination of such criteria has to be taken into account which leads to an optimization problem.
- *Knowledge base*: Then, in order to produce offers that are likely to fit a given utility function best, some kind of knowledge base is often employed which can contain either *domain knowledge* such as information about market values or *specific knowledge* about concrete negotiators obtained from previous encounters such as the result of the last negotiation on the same issue.
- *Protocol conformity*: There is a very close relationship between negotiation protocols and strategies. First of all, it is a necessary condition for every concrete strategy to compute offers or actions that conform with a given protocol.[1] Secondly, the protocol may have great influence on the efficiency of a strategy. For example, exploiting the *timeout* for a negotiation action specified by the protocol can be essential when negotiating with several parties, as shown in [16].

Classification Criteria In principle, a negotiation strategy can be realized by any algorithm that computes proper actions for a participant during the negotiation. And since a wide variety of possible algorithms, most of which are dedicated to some specific negotiation problem, has been proposed, it is not easy to classify them in an exhausting way. However, they can be grossly classified according to the following criteria:

- *mathematical/analytical*: are those strategies using some analytical method to compute negotiation actions.
- *heuristic/evolutionary*: are most strategies using some kind of *evolutionary programming* techniques.
- *local*: are strategies which do not depend on cooperation with other negotiators.
- *distributed*: On the contrary, distributed strategies make use of cooperation between negotiators.

The two pairs of contrasting criteria, i.e., analytical/evolutionary and local/distributed, thus yield two dimensions to characterize the strategies. In the next section, the differences between analytical and evolutionary strategies are briefly discussed.

[1] This condition does not apply, however, to meta-strategies (see 4.4).

Analytical versus Evolutionary Strategies Analytical and evolutionary strategies have both been often proposed, but are based on very different computing paradigms, each of which has some advantages as well as disadvantages in comparison to the other. Whereas the first employ some kind of relatively *static* mathematical model to compute negotiation actions, the latter make use of very *dynamic* computing techniques which are based on evolution principles such as selection, recombination and mutation (see [8] for an extensive treatment of evolutionary programming techniques).

Regarding analytical strategies, there already exist quite sophisticated and elaborated strategies for specific negotiation problems. For example, in [19], a technique of guessing the acceptance threshold of the other party (in a bilateral negotiation) based on the Bayesian method is presented. This technique also demonstrates that although the computing method is static, a learning effect can be achieved by using some knowledge base that is updated dynamically during the negotiation, so that every negotiation can take a different course. Analytical strategies have some advantages: They are immediately ready for operation and have a stable, reliable behavior. The main disadvantage is the potential predictability due to the underlying static model.

With evolutionary strategies, the learning effect is generally greater and also has a different dimension, since not only the data basis can evolve, but also the algorithms operating on these data themselves (which is called evolution based program induction). Thus, evolutionary strategies are principally much more creative and self-adaptable than those based on analytic models. However, there only exist a few implementations of simple, data oriented evolutionary negotiation strategies [13]. The main disadvantage of the evolutionary approach is that the resulting mechanisms always need a certain inititial phase to adapt so that they are not immediately ready for (effective) operation.

3 The DYNAMICS Architecture

The general goal of the DYNAMICS (DYNAMIcally Configurable Software) project is the design and implementation of highly configurable software components [6] which can be used as building blocks to assemble ready-to-use applications in a dynamic manner. With regard to mobile agents, this means that the functionality of an agent can be composed of several independently developed components which are plugged together. Especially, the agents should satisfy the requirements of role-specific functionality, on-the-fly loading and flexible configuration mentioned in section 1. In the following, some concrete components for building negotiating agents are introduced, which will then be described in more detail in section 4.

3.1 Components for Building Negotiating Agents

From an external point of view, negotiating agents are opaque entities which are just discernible by their exchange of negotiation messages. Different message exchange mechanisms can be used to distribute a message to one (unicast) or many (multicast) agents simultaneously, depending on the underlying negotiation protocol.

As illustrated in Figure 1, however, a negotiation enabled agent in the DYNAMICS architecture is internally structured into the following main components:

- *Communication module (C)*: This component is concerned with the delivery and processing of any kind of messages exchanged between the agents (see 4.1). When a message is recognized as a negotiation message, its content is passed to the protocol module.
- *Protocol module (P)*: This is the component responsible for the protocol compliance of an agent, which implies that the content of each incoming and outgoing negotiation message is inspected by the protocol module. It can be implemented either as an independent entity or as a front-end of a central protocol engine which seems more appropriate in case of protocols with complex semantics (see 4.2).
- *Strategy module (S)*: This is the component that implements a negotiation strategy which is responsible for producing proper negotiation actions as required by the protocol module. This module can also directly call the communication module, as in case of distributed strategies (see 4.3).

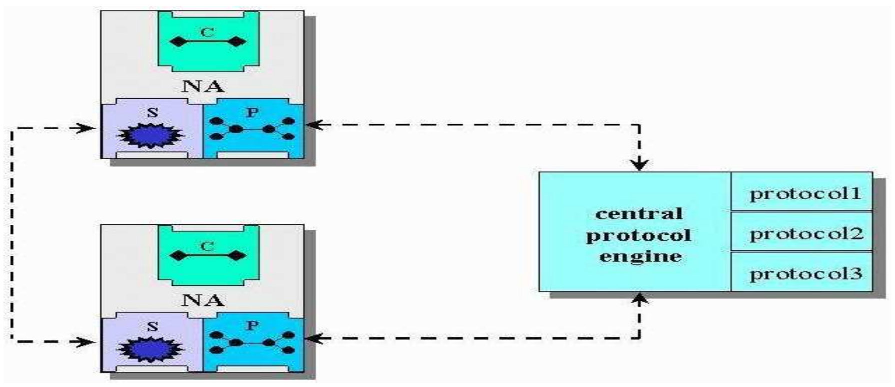

Fig. 1. Main components of negotiation enabled agents

This modular structuring has some obvious advantages: The functionality of each module is clearly defined by interfaces so that it can be developed independently using very different implementation methods or algorithms, as in case of the strategy module. Moreover, a component such as the protocol module can be provided and/or certified by a third-party instance in order to prove the correct behavior of an agent with respect to a protocol. In this way, a clear separation of “private” (strategy) and “public” (protocol) matters is achieved.

3.2 Plug-in Types

In the DYNAMICS architecture, most of the application semantics of mobile agents is realized by *plug-ins* which are components that can be added to and removed from agents at run-time. This means that in the first place, these agents can be seen as plug-in containers which provide a minimal, orthogonal functionality of mobile agents, i.e. mobility and persistence. Plug-in components, which can be dynamically incorporated into agents to provide application semantics, are classified into the following types:

– *Roles*: Roles are plug-ins that introduce new functionality into agents or entities which serve as role containers. Intuitively, the functionality associated with a role represents the semantics of some business entity such as "seller", "buyer" or "notary". Adding a role to a plug-in container object means providing this object both with a new (or additional) interface and a corresponding implementation (which may be loaded on demand).
– *Substitutes*: A substitute is a plug-in that is used to provide a new or replace an existing implementation for some interface, i.e. the interface remains unchanged.
– *Configurations*: A configuration plug-in is used to reconfigure an application component dynamically. That means, both the interface and implementation of the reconfigured component remain unchanged. Typical of a configuration plug-in is the rule module (R) (see 4.4) which can be used to impose a constraint, for example, about the total budget available for the negotiation, on the strategy module. Adding such a rule module results in the plug-in structure of negotiation enabled agents depicted in Figure 2.

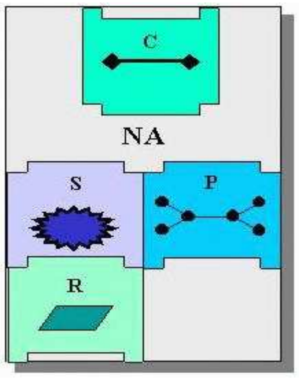

Fig. 2. Completed plug-in structure of a negotiation enabled agent

4 Negotiation Support Building Blocks

In this section, we describe the functionality and outline the design of the main modules serving as building blocks for negotiation enabled agents.

4.1 Communication Module

The communication module is a role plug-in that provides an agent with the capability of using a communication language, or to put it figuratively, it transmutes a (basic) agent into a "speaking" agent. In order to be interoperable with other agent systems as much as possible, we have chosen to use KQML [3] for the communication module. KQML, which was developed as part of the DARPA Knowledge Sharing Effort [14], offers the following advantages:

- Commonly recognized standard for agent communication.
- Enabling flexible, asynchronous exchange of informations and action requests.
- Applicable for many different application fields by introducing an *ontology* for each field (e.g., medicine).

The communication module is itself a plug-in container in which several *message interpreters* can be dynamically embedded to deal with different ontologies. In order to identify negotiation messages, a new ontology *negotiation* has been introduced. That means, when an incoming message is recognized as belonging to the ontology negotiation, its *content* field is passed to the corresponding interpreter which performs a syntax check and then passes it further to the protocol module.

4.2 Protocol Module

The protocol module is a substitute plug-in called by the communication module, or more precisely, by the corresponding interpreter embedded in the communication module. The interface between these modules is basically determined by the following parameters:

- *In*: Incoming negotiation message content and sender of message.
- *Out*: Outgoing negotiation message content, addressee(s) of message and/or sending mode (unicast/multicast).

In case of simple negotiation protocols, which do not require any essential coordination and synchronization between the participants, the protocol module can be implemented as an independent component which is completely loaded into the agent. For example, an agent participating as a prospective buyer in an auction needs with regard to the protocol only to determine whether he is treated correctly by the auctioneer, i.e. make sure that he is sold the good if he made the highest last bid.

However, in case of protocols with complex semantics (see, e.g., the 3-peer scenario described in [5]), it is more appropriate to provide the protocol module as a front-end to a central protocol engine (as depicted in Figure 1). Such a central engine is a specialized workflow engine executing protocols specified in a Petri-net based workflow description language (see [10]) which is suitable to express the dynamic aspects of a negotiation process described in section 2.1.

4.3 Strategy Module

The strategy module is a substitute plug-in called by the protocol module. The strong protocol dependence described in 2.2 entails that the interface between the strategy and the protocol module cannot be statically specified, i.e. using static data types, but has to allow dynamically typed parameters:

- *In*: The current negotiation state is passed from the protocol to the strategy module. What constitutes the negotiation state is mainly dependent on the protocol.[2] Optionally, the protocol module can pass the set of all protocol compliant actions, from which the strategy module can choose one to proceed with, as in case of the contract net protocol ([2]).
- *Out*: The negotiation action computed by the strategy module, for example, a counter-offer to the last offer. Which offers are valid is also dependent on the protocol.

Optionally, the strategy module can have interfaces to additional components such as a knowledge base (see 2.2) or it can make use of the communication module to interwork with other strategy modules in case of distributed strategies.

4.4 Rule Module

The rule module is a configuration plug-in for the strategy module. Generally, rules can be seen as objects that do not provide an external functionality for the components they are plugged into through call interfaces, but rather (re-)configure or constrain these components. The configuration effect can be achieved by performing actions which manipulate the (external) properties of the configured component, as shown in [18].

Rule Types Rules in the DYNAMICS architecture are classified into the following types:

- *Invariants*: An invariant is a condition that must hold true at any time with regard to the entity it is assigned to, which can be an agent or a group of (cooperating) agents. For example, the invariant `(budget` $>$ `0)` $\wedge$ `(size` $\leq$ `100)` can be interpreted as: The budget the agent may use to negotiate must be always positive and its total size must not exceed 100 units.

[2] In an auction, for instance, it could be just the last bid, but for another protocol, it might be a vector containing the negotiation states of all participants.

- *Policies*: A policy consists of a condition, called *goal*, and an action which leads to the goal, when it does not (longer) hold. For example, the policy
  P1: `(budget` $\geq$ `100)` $\leftarrow$[3] `deposit(1000)`
  specifies that the action deposit(1000) is to be performed when the budget becomes less than 100 units.
- *Action rules*: An action rule consists of a condition, called *trigger*, and an action to be performed when the trigger holds. For example, the action rule
  A1: `(budget` $<$ `100)` $\rightarrow$ `deposit(10)`
  specifies that the action deposit(10) is to be performed when the budget becomes less than 100 units.[4]

Thus, invariants can be seen as *passive* rules, whereas policies and action rules are *active* ones. See [18] for a detailed description of mechanisms to formalize, evaluate, unify, compare and activate policies, most of which are applicable to the other rule types as well.

Using Rules to Implement Meta-Strategies Using rules to configure the strategy module can achieve the effect of *general* strategies, i.e. those that are applicable for many negotiation types and in particular are not dependent on a concrete negotiation protocol. For example, the space between two price offers can be specified by a policy which is manually specified or computed by a *meta-strategy* component which is protocol independent. In this way, simple general behavior patterns expressed in terms of "cautious/patient" or "risky/fast-paced" can be easily imposed on the strategy module and such a meta-strategy component can be reused for different negotiation protocols and strategies.

5 Implementation Issues

An implementation which is so "dynamic" that it allows for the compositional substitution as required by the concept of roles outlined above is quite demanding on the system technology's capabilities. Separating interfaces from their implementations, the possibility to substitute both of them at run–time as well as representing and evaluating domain knowledge supporting an agent's strategic decisions are the main challenges.

Choosing typical object–oriented, class–based "production" languages such as C++ or Java for realizing the required plug–in architecture leads to the

[3] Please note that in this context, "$\leftarrow$" and "$\rightarrow$" do not denote *logical implication*, but rather serve as symbols for the "causal" relationship between a condition on the one hand and an action on the other hand.

[4] Although policies and action rules seem to have similar semantics, they are not interchangeable, since the action of a policy must lead to the goal (otherwise, an exception will be thrown), whereas the action in an action rule can be any arbitrary one. To give another example, the expression A2: `(budget` $<$ `100)` $\rightarrow$ `dieNow()` represents a correct action rule, whereas P2: `(budget` $\geq$ `100)` $\leftarrow$ `dieNow()` does not represent its policy counterpart.

well–known problem of *class evolution* of systems which treat classes as (static) compile–time concepts. On the other hand choosing possibly more appropriate environments like typical interpreted AI languages as Scheme or CLOS which allow for more dynamic, (self–) modifiable systems as well as proven knowledge representation techniques lacks the ubiquitous availability required by a mobile agent system.

Therefore, the "pragmatic" decision for the DYNAMICS project was to build the whole architecture on top of the widespread Java–Technology available with minimal effort as an ubiquitous networked infrastructure. The first prototype of the here presented *pluggable agents* architecture has been implemented on top of Objectspace's Java–based *Voyager* system[5] [12], which provides the basic functionality for building mobile agents in a very efficient manner. The decision has been not to choose one of the "main" agent systems like Odyssey [4] or Aglets [7] but an infrastructure that provides a small, clear concept of movable objects, thus keeping unnecessary overhead small, but nevertheless having a solid base to build the specially required agent containers on. Also, Voyager's small footprint is well suited for ubiquitous network distribution.

The dynamic pluggability of the components which can be used to assemble a role-specific agent is based on the generic concept of a *Pluggable* which is implemented using the *MessageEvent* mechanism of Voyager to delegate method calls to the right target(s). The design of the main interfaces and classes implementing this plug-in mechanism is depicted in Figure 3.

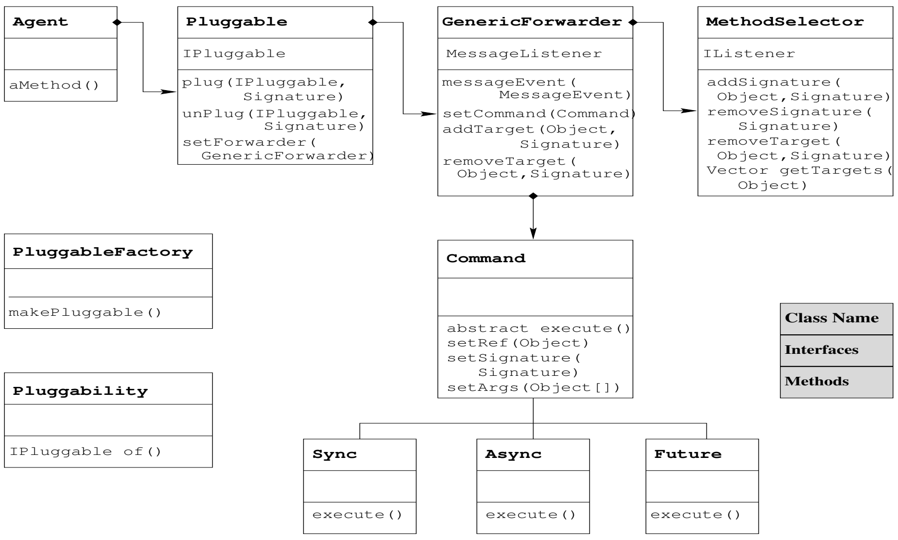

Fig. 3. Class diagram for plug-in mechanism

[5] The version currently being used for our prototype implementation is 2.0 Beta 2

`IPluggable` is the interface common to all objects that can act as a plug-in container by providing the methods `plug` and `unPlug` to add a plug-in (called *destination plug*) into or remove it from the container object (or *source plug*). `Pluggable` is one implementation of the `IPluggable` interface using a generic forwarder, which can be dynamically set by the method `setForwarder`, to delegate method calls to their target objects. The `GenericForwarder` implements Voyager's `MessageListener` interface, defines methods for adding and removing targets of requests (`addTarget` and `removeTarget`), provides the forwarding mechanism through the method `messageEvent` and enables different call semantics through the method `setCommand`. To handle the message events, the generic forwarder makes use of the class `MethodSelector` which defines methods for filtering message events based on signatures (methods `addSignature`, `removeSignature`, `select`) and for determining the targets of the signatures (methods `getTargets`, `removeTarget`). The abstract class `Command` provides a generic DII mechanism for executing method calls on targets (through `execute`) and defines methods for dynamically changing targets, method name and parameters to method calls. `Sync`, `Async` and `Future` are implementations of `Command` corresponding to the call semantics *synchronous*, *asynchronous* and *future* in Voyager, respectively. The `PluggableFactory` class provides static factory methods in order to construct plug-ins at run-time. `Pluggability` provides a static method (`of`) in order to add the plug-in capability to an object dynamically. Finally, `Agent` is a plain Voyager agent inheriting from `Pluggable` to serve as a plug-in container and containing some (application-specific) code to administer and manipulate plug-ins based on the `IPluggable` interface. For example, such an agent can have some code to determine if or when the plug-ins should migrate with it (or when to move somewhere else). In summary, the plug-in mechanism outlined above

- facilitates communication and cooperation between software components by establishing a unidirectional request forwarding mechanism from source plug to destination plug.
- allows plug-ins to register a method signature with the source plug which is used for efficient event filtering and method selection in the source plug.
- enables the use of different call semantics in order to express the execution dynamics of different scenarios more adaquately.
- decouples source plug and destination plug by indirecting the cooperation mechanism through a more general message listening mechanism.

Using this dynamic plug-in framework, a set of communication modules (see Section 4.1) including a KQML plug–in and rule modules supported by the policy management system described in [18] have been implemented enabling agents to switch their application level communication and changing their configurations (state, properties) according to the tasks they are expected to do. At present, a couple of prototype (negotiation) protocol and strategy modules are under development giving the agents the possibilty to act in different "market scenarios" like auctions, wholesaling, black–boards and flea–markets adapting to the varying behavior and strategies found in these scenarios.

6 Summary and Outlook

In this paper, we have presented an approach of dynamically embedding negotiation capabilities into mobile agents. First, it was shown that the requirements for enabling automatic negotiations, with respect to both negotiation protocols and strategies, are manifold, the main consequence of which is that a framework of corresponding building blocks has to be generic and flexible enough to be able to support a wide variety of protocols and strategies. Then, a plug-in architecture for mobile agents consisting of four main negotiation support modules and the corresponding plug-in types required to implement them was proposed. Next, the design of the concrete modules was presented and finally, some relevant implementation issues were described.

However, there are some issues which have not been addressed in this paper. The first one concerns the question of how the agents can find appropriate partners to start a negotiation in an open environment such as the Internet without being given prior knowledge. Related to this is the question of how to establish a group or consortium of partners which can participate in a negotiation as one role. Another issue is how to provide as much support as possible for the execution of a negotiated result or contract. These are some questions which we are currently examining in the context of a generic contracting service for electronic commerce applications.

Acknowledgement This work is supported, in part, by grant no. La1061/1-1 from the German Research Council (Deutsche Forschungsgemeinschaft, DFG).

References

1. D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant agents for mobile computing. Technical Report RC 20010, IBM Research Division, T.J. Watson Research Center, 1995.
2. R. Davis and R.G. Smith. Negotiation as a Metaphor for Distributed Problem Solving. *Artificial Intelligence*, (20):63–109, 1983.
3. T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94)*. ACM Press, November 1994.
4. General Magic. Odyssey, 1997. www.genmagic.com/agents/.
5. F. Griffel, T. Tu, M. Mnke, M. Merz, W. Lamersdorf, and M. M. da Silva. Electronic Contract Negotiation as an Application Niche for Mobile Agents. In *Proceedings of the First International Wokshop on Enterprise Distributed Object Computing, EDOC'97, Australia*, pages 354–365. IEEE, Oktober 1997.
6. Frank Griffel. *Componentware.* dpunkt–Verlag, 1998. (In German).
7. IBM. Aglets, 1997. www.tri.ibm.co.jp/aglets/.
8. C. Jacob. *Principia Evolvica : Simulierte Evolution mit Mathematica.* dpunkt–Verlag, 1997. (In German).
9. Boris Liberman, Frank Griffel, Michael Merz, and Winfried Lamersdorf. Java–Based Mobile Agents — How to Migrate, Persist, and Interact on Electronic Service

Markets. In Kurt Rothermel and Radu Popescu-Zeletin, editors, *Proceedings of the First International Workshop on Mobile Agents, MA '97, Berlin, Germany*, number 1219 in LNCS, pages 27–38. Springer, April 1997.

10. K. Müller–Jones, M. Merz, and W. Lamersdorf. Realisierung von Kooperationsanwendungen auf der Basis erweiterter Diensttypbeschreibungen. In H. Krumm, editor, *Entwicklung und Management verteilter Anwendungssysteme — Tagungsband des 2. Arbeitstreffens der GI/ITG Fachgruppe 'Kommunikation und verteilte Systeme' und der GI Fachgruppe 'Betriebssysteme'*, pages 20–30. Universität Dortmund, Oktober 1995. (In German).
11. J.P. Müller. A Cooperation Model for Autonomous Agents. In J.P. Müller, M.J. Wooldridge, and N.R. Jennings, editors, *Intelligent Agents III: Agent Theories, Architectures, and Languages (Proceedings of ECAI'96)*, LNCS. Springer, August 1996.
12. ObjectSpace. Voyager — core technology user guide, Dezember 1997. `www.objectspace.com/voyager/documentation.html`.
13. Jim R. Oliver. *On Artificial Agents for Negotiation in Electronic Commerce.* PhD thesis, Wharton, 1996. `wharton.upenn.edu/~oliver27/dissertation/diss.zip`.
14. R. Patil, R. Fikes, P. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches. The DARPA Knowledge Sharing Effort: Progress report. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Third International Conference (KR'92).* Morgan Kaufmann, November 1992.
15. J. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiations among Computers.* MIT Press, 1994.
16. T. Sandholm and V. Lesser. Issues in Automated Negotiation and Electronic Commerce: Extending the Contract Net Framework. In V. Lesser, editor, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 328–335, San Francisco, June 1995. AAAI / MIT Press.
17. M. Stra*ß*er, J. Baumann, and F. Hohl. Mole - A Java Based Mobile Agent System. In *Special Issues in Object-Oriented Programming, Workshop Reader ECOOP'96*, pages 327–334. dpunkt–Verlag, 1996.
18. M.T. Tu, F. Griffel, M. Merz, and W. Lamersdorf. Generic Policy Management for Open Service Markets. In H. König and K. Geihs, editors, *Proc. of the Int. Working Conference on Distributed Applications and Interoperable Systems (DAIS'97), Cottbus, Germany.* Chapman & Hall, September 1997.
19. D. Zeng and K. Sycara. How Can an Agent Learn to Negotiate? In J.P. Müller, M.J. Wooldridge, and N.R. Jennings, editors, *Intelligent Agents III: Agent Theories, Architectures, and Languages (Proceedings of ECAI'96)*, LNCS. Springer, August 1996.