A Dynamic Negotiation Framework for Mobile Agents

M.T. Tu, C. Seebode, W. Lamersdorf *
Distributed Systems Group, Computer Science Department, University of Hamburg, Germany
Vogt–Kölln–Str. 30, 22527 Hamburg, Germany
[tu|1seebode|lamersd]@informatik.uni-hamburg.de

Abstract

In this paper, a framework to integrate negotiation capabilities – particularly components implementing a negotiation strategy – into mobile agents is described. This approach is *conceptually* based on the notion of an *actor* system which decomposes an application component into autonomously executing subcomponents cooperating with each other. *Technically*, the framework is based on a plug-in mechanism enabling a dynamic composition of negotiating agents which is presented as a complete design pattern.

Keywords: negotiation, mobile agent, interaction patterns, electronic commerce.

1 Introduction

Deploying agent technology – especially mobile agents – as a basis for building the information infrastructure of an emerging "Net"-society is one of the most challenging issues in many research areas. Speaking drastically, one can think of this kind of research, which is more and more associated with notions like "community computing" [Ish98], as designing the future society. This is at least true for the deployment of mobile agents in electronic commerce. Being the central interaction scheme in economic activities, negotiation is one of the main focuses in the development of software agents to perform online commercial transactions. The incorporation of electronic commerce capabilities into mobile agents has been developed under several aspects [VMK98, RMdS98]. However, the integration of intelligent capabilities like those needed for negotiation into mobile agents raises new requirements leading to a different research path, which focuses on integration issues that can be dealt with on a general level, i.e. independent of the respective contributing research areas [TGML98].

This paper presents a component architecture of self-interested negotiating mobile agents. It is implemented in the context of the DynamiCS[1] project at the University of Hamburg. This architecture puts a strong emphasis upon the fact that mobility and intelligence are not opposed, but rather orthogonal to one another. The corresponding integration of these two aspects is technically based on a dynamic *plug-in mechanism* which supports the assignment of desired capabilities at runtime. In this architecture, the *intelligent* capabilities that are necessary to enable a certain type of negotiation (at a time) are further divided into three main categories:

- Communication
- Negotiation protocol compliance
- Strategic thinking.

*This work is supported, in part, by grant no. La1061/1-2 from the German Research Council (Deutsche Forschungsgemeinschaft, DFG)

[1] *Dynamic*ally *C*onfigurable *S*oftware

Each of these aspects is modeled as an individual *component* which represents a set of features that can be dynamically specified. This architecture reflects the agent then as a collection of such individual components. Of course, the negotiation performance or overall intelligence of such an agent depends heavily on its *composition*, i.e. on the combination of the respective negotiation strategy and protocol. However, it is a central feature of the presented architecture that the choice of strategy, protocol or communication language is not restricted by any technical issues which arise in the context of integration. On the contrary, it is an explicit goal to subsume different kinds of intelligent capabilities into the same architecture and to make it possible to switch between them dynamically, even in the same negotiation. Whereas the issues of communication language and protocol compliance are also discussed in other publications [TGML98, TLGL99], this paper will have a specific focus on how to embed different negotiation strategies into mobile agents and the corresponding plug-in mechanism.

The remainder of the paper is organized as follows:

- Section 2 describes an actor-based negotiation framework capturing the functional decomposition of the mental capabilities of a negotiating agent into active objects. In particular, the structure of this framework, its main properties and a practical application approach are presented.
- Section 3 presents a dynamic plug-in mechanism as the basic composition technique of the framework. The conception of this technique is described in terms of a design pattern including its concrete system design and the current state of the prototype implementation.
- Section 4 finally sums up the paper by giving an outlook on current work done in the project and some open issues that need to be investigated further.

Implementation constraints

The agent architecture presented here is embedded into the DynamiCS project at the Distributed Systems Group at University of Hamburg. For implementation work related to this project, Java [Mica] was chosen as the implementation language and Voyager [Obj] as the basic mechanism for distribution and mobility. However, the aim of this architecture is to study the basic requirements of a system of self-interested negotiating mobile agents which can be considered independently of any concrete implementation.

2 An actor-based negotiation framework

The development of self-interested negotiating agents requires an understanding of the sequence of events in an negotiation. The structuring of the overall task of an negotiating agent into specialized *modules* is the main design rationale behind the construction of the DynamiCS agents.

Modules represent the agent's capability to

- **communicate** in different control languages (e.g., KQML or XML) following either a stream-oriented communication model or to expose an object-oriented communication interface that other agents holding a reference can use to post their messages.
- **comply with negotiation protocols** in order to take different roles in negotiations or to detect protocol incompliant behavior of other participants.
- **think strategically** to maximize the benefit of the negotiation for the agent.

This modular approach allows for encapsulating the complexity of each task. Communication and protocol capabilities are enforced by environmental requirements whereas the choice of strategy relates to the self-interest of the agent. The choice of strategy is what decisively contributes to the success of a negotiating agent. The implementation of a negotiation strategy realizes a more or less sophisticated model of the agent's intelligence concerning this goal.

A negotiation strategy in general is a mapping between a sequence of negotiation messages (the negotiation history) to a set of possible actions (determined by the specific protocol) taken in response. Since there are many different possible approaches for the implementation of negotiation strategies, a classification helps to understand the major differences. Negotiation strategies can be divided into analytical or genetic/evolutionary techniques differing principally with respect to the learning effect that can be achieved. In the former case, some fixed algorithm is employed to compute the actions whereas in the latter case, an essential part of the algorithm or strategy itself undergoes dynamic modifications based on evolutionary principles – such as mutation, crossover and recombination – and thus, the strategy literally evolves over time [Oli96]. A different classification scheme examines whether all application logic is executed locally with respect to the agent's boundaries or if there is a coordination between distributed entities contributing to the decision finding [TGML98].

Building a framework for the development of negotiation strategies requires to respect at least two different viewpoints. First is the *developer's viewpoint*. The developer of a concrete strategy needs support for the integration of his design into the general architecture of an agent. A developer uses a concrete model of a problem domain, builds data structures and algorithms that perform the evaluation of a negotiation situation.

The *framework's viewpoint* is quite different in this sense. A framework for the development must not impose any restriction on the domain model whatsoever. The framework is concerned with the delivering of negotiation messages to the domain model and converting the evaluation back into actions taken (the response messages) by the agent.

This shifts the framework's viewpoint from a domain model to an *execution* model. The execution model has to support the execution of the desired task and all resource monitoring necessary for the agent's performance in a dynamic environment. The heterogeneous nature of the agent's actions is a challenging task for the design of an execution model. On the one hand, an agent plays an autonomous, proactive role by issuing negotiation messages to other agents. This happens for instance in an auction scenario, where any agent can deliberately posts bids. On the other hand, an agent plays a reactive role when responding to messages generated by other agents.

2.1 The structure of the framework

A combination of proactive and reactive behavior in an execution model requires non-blocking communication between the participating components. The general execution model of the framework therefore models a negotiation strategy as a hierachical set of active objects which corresponds to the notion of an ACTOR system [Agh86, Hew77]. An *actor* executes in response to message passing and performs atomic instructions that execute as a whole and in concurrency to other actors. A single instruction can be part of an algorithm or a message gateway to another agent. However, the framework does not prescribe the concrete number of and relationships between actors, but only requires that they be controlled by means of a *coordinator* which also represents the external interface of the whole strategy module.

Coordinators can be modeled as special actors called *synchronizers* [Frø96] that control message dispatch to a group of actors (see Figure 1). Synchronizers constrain the execution of their controlled actors by defining a set of rules and evaluate them before dispatching messages to a group of actors.

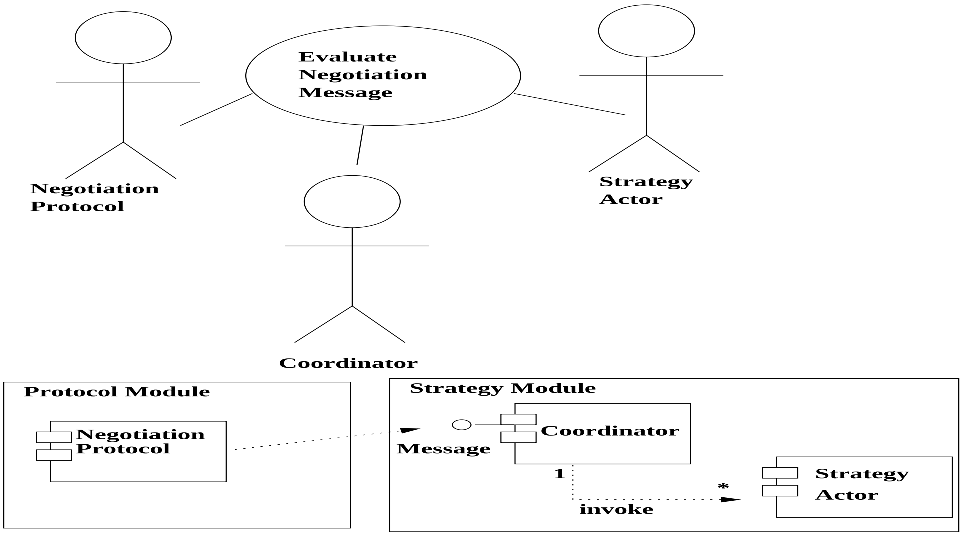

Figure 1: Roles and their distribution in the negotiation framework

With coordinators and actors as roles in the execution model, a *hierarchical* composition of strategies is possible (see Figure 4). Even the integration of different classes of negotiation strategies, e.g. a combination of analytical and evolutionary algorithms, into one agent is supported. The developer's task is to adapt the domain model to the actor model.

The framework takes care of delivering all the negotiation messages included into the negotiation protocol to the strategy. The integration of a negotiation strategy means to specify

- the atomic instructions to be executed by the actors
- the conditions that constrain the actors' actions
- the messages of interest for the strategy.

One possible example of mapping actors to atomic actions could be to instantiate several different actors to calculate a possible response concurrently. The coordinator calculates the utility of each response delivered by an actor and then selects the response with the highest utility value for the agent.

2.2 Framework properties

A framework supporting an execution model of dynamic negotiation strategies is able to introduce fine-grained control of the agent's execution. With the possibility to execute tasks in parallel by delegating them to the actor system, there are different kinds of constraints to be considered with respect to the control of the executable tasks. In the DynamiCS architecture, the execution control of the actor system is delegated to the coordinator role. The coordinator checks the validity of execution constraints. Constraints can be classified according to different levels of execution as follows:

- strategy constraints. Constraints that control the execution of actors that model an negotiation strategy (i.e. the control structure of the underlying algorithm).
- agent constraints. Constraints that reflect the inner state of the agent (i.e. checking if the agent is preparing to migrate).
- negotiation constraints. Constraints that reflect the state of a negotiation (i.e. checking if running evaluations are still consistent with the state of an negotiation which is determined by the respective negotiation protocol).

From the framework's viewpoint, the negotiation constraints are certainly of highest interest because the framework can be seen as providing the structure of *negotiation-enabled* agents. Since negotiation is in most cases a very dynamic process during which a participant has to be able to react to relevant events occurring at *any time*, such as a new offer made by another participant, and since there is generally a trade-off between computation time and quality (w.r.t. some utility function) of computed negotiation actions (comparable to many games), it is a very desirable feature that the algorithms underlying a concrete strategy can be interrupted at any point of computation and nevertheless delivering some usable result. In this respect, the execution model proposed here is consistent with the demand for *anytime properties* (as proposed by [Zil96]) of the tasks carried out by the agent. The central issue in anytime computing as in other resource-bounded computing techniques is the explicit control of meta-information concerning the computational resources.

Especially, the control of *quality* and *duration* as claimed by progressive processing techniques [MZD98] can be supported by the model. Actors and coordinators together are the toolkit to establish an explicit progression in computing time and quality. These concepts can easily be modeled as parameters of an actor system.

2.3 A sample application: applying genetic algorithms to the framework

In this section, in order to demonstrate how the inherent concurrency of the proposed architecture can be exploited to enhance the performance of existing negotiation strategies, we will present an approach of integrating genetic algorithms into the strategy framework. The simple genetic algorithms deployed here are based on the work described in [Oli96] and basically function as follows:

Strategies are modeled as simple sequential threshold rules made up of offers separated by thresholds which represent the total utility value of an offer. Offers themselves are modeled as tuples of values corresponding to negotiable attributes (e.g., price, quality, delivery etc.) each of which has a certain utility value (see Figure 2).

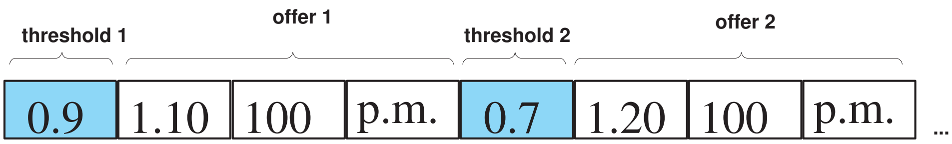

Figure 2: Numerical example of an evolution-based strategy (following [Oli96])

System operation begins with building a population of random strategies for the agent which successively takes a strategy at a time to take part in a negotiation and calculates its payoff when the negotiation ends. After such a population has been tested in this manner, a new one is produced by selecting the strategies with the best payoffs and applying genetic operators such as mutation and crossover on them to generate new ones filling the new population. When this process is iterated a number of times using different agents to play against each other, a certain learning effect is achieved.

With respect to the presented actor-based framework, this simple method can be modeled as a strategy module containing the coordinator and only a single actor which returns the next element of the offer sequence in each atomic instruction (see Figure 3a). However, it can easily be seen that the performance of the overall genetic algorithm can be improved by using several actors concurrently (see Figure 3b), even when the coordinator just successively (or randomly) takes one output of the actors to perform the negotiation. Of course, in the first round, everything is the same as with one single actor since the offer sequences are generated randomly, but from the second round on, the learning effects of several single strategies are accumulated.

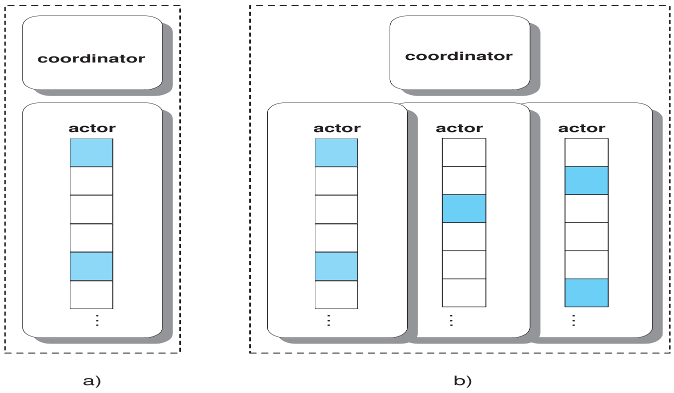

Figure 3: Integrating evolution-based strategies into the framework

Moreover, this method can also be enhanced by using more than one level in the hierachy of coordinator-actor with each actor on one level being the coordinator on the next level except for the last one. Within each level, the coordinator can then exploit the available execution time to train the actors by performing test negotiations (i.e. simulations) until a certain *fitness* has been reached or the available time is over. Figure 4 illustrates this technique for two levels.

Also, with respect to the anytime properties mentioned above, it can easily be shown that such a hierachical genetic algorithm has the desired features of:

- Interruptibility: Meaning that the algorithm can be stopped at any time providing some answer. This is true because the actors on the last level can be stopped at any time providing the offers just taken out of the respective offer sequences. By means of the coordinator on the same level, one of the results can be selected and forwarded to the next higher level and so on up to the highest one so that the algorithm as a whole is interruptible.

- Recognizable quality: Meaning the quality of an approximate result can determined at run time. This is true because after every round of (real or simulated) negotiation, the fitness of the applied strategies can be determined

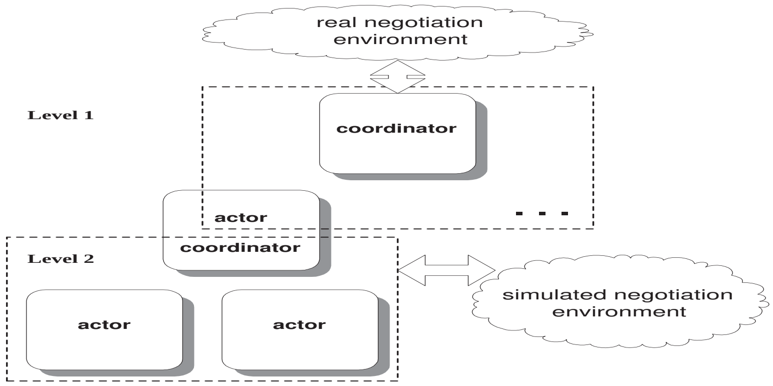

Figure 4: Structure of a hierachical strategy with 2 coordinator-actor levels

by predefined functions. (In the first round, they are assumed to have the same fitness.)

- Monotonicity: Meaning the quality of the result is a nondecreasing function of time and input quality. As pointed out in [Zil96], this can be guaranteed when recognizable quality is ensured, simply by returning the best result generated so far.

3 A dynamic plug-in mechanism

This section presents the basic composition technique of the framework. The modular architecture presented so far decomposes into a hierarchy of different abstractions of what a negotiating agent is supposed to do. On a coarse scale, the agent consists of modules, which are selected and composed dynamically reflecting the requirements of a constantly evolving environment. On a finer scale, each module – especially the strategy module presented above – decomposes into a set of tasks, contributing to the module's overall goal, which can be dynamically assigned to active objects (*actors*). No matter which abstraction level is considered, the need for dynamic composition of the participating entities is evident. So how is this dynamic composition achieved?

3.1 The Plug-In mechanism as a Design Pattern

A very flexible way of dynamic composition is to introduce runtime relations between components that were not known at compilation time. We call this a plug-in mechanism, because it emphasizes the notion of a plug having a well-known coupling interface which establishes the relation. Defining a general plug-in mechanism apart from the type of information that flows through this link is a powerful concept in dynamically evolving environments. A plug-in mechanism models a cooperation between components. It allows to declaratively specify two important concerns of cooperation [Agh97]:

- *What* is going to cooperate and
- *When* is it going to cooperate.

Breaking this concept down to the implementation level, cooperation between object-based software components means to establish a relation between the method calls of these components. Two dimensions can be identified for specifying such a relation. They basically describe whether the cooperating methods execute in parallel or in serial and if the there is an parameter dependency to be established between the two methods. The following table documents the combinatorial manifoldness of this relation, and shows how each case can be modeled by the cooperation mechanism.

	parallel	serial
parameter dep. +	Notification on enter input parameter forwarding	Notification on leave input or output parameter forwarding
parameter dep. -	Notification and call forwarding without parameter	

Table 1: Possible cooperation patterns

Object-based software components expose the information needed for this relation at the public interface. Our plug-in mechanism is designed to intercept the message flow through this interface and to forward it to the cooperating component (i.e. the *target plug*). The plug-in mechanism is responsible for the forwarding of a message sent to a source component to the target components that are registered for the corresponding message event. This models an asymmetric relationship between components, since with respect to the message forwarding mechanism, there are *source* components and *target* components. This is frequently expressed by publish/subscribe interfaces. Events flowing through these interfaces can have static types or a generic type. The dynamics of negotiating agents makes it difficult to design for statically typed messages. This is why a message forwarding mechanism for negotiating agents should provide a declarative way to specify message types, that can be evaluated at runtime. Another design issue is the asymmetrical relationship between Publisher (the issuer of the event) and Subscriber (the receiver of the event). Even if this asymmetric relationship between source and target components can be identified in principle, coding this asymmetric relationship of two components into separate interface is a design time issue. In order to support a possibility to dynamically assign the pluggability feature to a component, it's easier not to separate the concerns of publisher and subscriber. At runtime one component can play different roles in different cooperation scenarios. Hence the plug-in mechanism only defines *pluggable* components which can cooperate in any direction. Only the cooperation pattern has to be specified explicitly. The cooperation pattern is a declarative way to specify methods and parameters for components that cooperate. The plug-in mechanism uses this cooperation pattern for the correct call forwarding and conversion of parameter lists if necessary. The cooperation pattern contains basically the data needed for the configuration of a dynamic invocation interface. Moreover, another requirement for dynamics results from the desire to be able to assign the plug-in capability itself (represented by the class *Pluggability*, see 3.2) to any component, which has not been programmed for this purpose, at run-time. This requires that the plug-in mechanism not only contains the logic for notification on message events, but also the possibility to enforce the consequences of such events.

The possibility to provide dynamic extensions to existing components demonstrates a way to establish the *Open/Closed Principle* [Mey88] to component-based software. Meyer's original discussion on the Open/Closed principle referred to modules as compilation units, but the plug-in mechanism refers to modules as runtime components. They are closed in the sense of providing a stable reference which is used at runtime for accessing all public methods. They are open for extension in the way the plug-in mechanism establishes a runtime relation between method invocations in cooperating components.

We think that if the plug-in mechanism is used as a general way for dynamic coupling of software components, than it is best described as a design pattern using the format proposed in [GHJV95][2]. However, although this pattern is a valid description of the components involved in the plug-in mechanism, its implementation relies on two basic services:

- A so-called Message Listening service which is a facility to provide notifications of the method calls destined for a certain component.
- A service to perform dynamic invocations on components with different call semantics (synchronous, asynchronous).

In our implementation, these services are provided by ObjectSpace's Voyager Framework [Obj].

3.2 Description of the Plug-In pattern

Intent

Describe a basic pattern for the cooperation of object-based components, by defining cooperation as a coupling between method calls in different components. Provide a mechanism for dynamically changing the behavior of software systems by assigning pluggability dynamically to components, and to make them cooperate through this mechanism.

Motivation

In a negotiation scenario, an agent tries to evaluate the negotiation history in order to supply an optimal offer, considering his own goals. The current strategy adapts the agent's knowledge, expressed as formal rules (e.g. "don't overspend"), to the current negotiation state. An intelligent agent may have a notion of its own performance, in terms of achieving the goals. It may be necessary to switch the strategy because the agent performs suboptimally (w.r.t. a certain negotiation goal).

The new strategy is dynamically loaded and plugged into the agent. This plugging involves dynamic invocations of the new features loaded. The plug-in pattern supports this dynamic coupling without explicit knowledge or referencing between components.

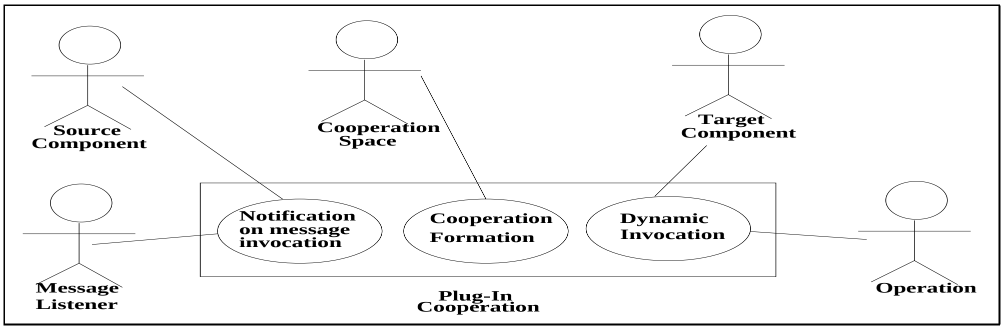

Figure 5: Use Cases in a cooperation scenario

[2]There are other templates for pattern description, see [MD98] for a discussion.

The plug-in pattern decomposes a *cooperation scenario* between two components into three complementary actions (*Figure 5*):

- Notification on message invocation
- Cooperation formation
- Dynamic invocation

This becomes less abstract considering a real invocation scenario. Basically, a cooperation between components can be described as a cooperation between methods of these components. A source method triggers the dynamic invocation of a target method. The triggering relies on a notification of method invocation. The target method is addressed by the description of its signature and parameters. After the selection of target methods, these are executed by a dynamic invocation.

The dynamic loading and linking of components often requires components that have been designed for that purpose. In order to support a more dynamic scenario, it is helpful to express the loading and linking information more declaratively. This makes it possible to assign the pluggability feature dynamically at runtime. In consequence, an agent is a collection of plugs which are plugged into one another. Although the message flow is directed in a particular invocation scenario, every plug can have multilateral relations, being either receiver or sender of messages.

Applicability

Use the plug-in pattern when you want to dynamically modify the cooperation capability of component based software systems. It can be used for components that were developed without knowledge of the plug-in mechanism. It is based on top of a method notification mechanism and a dynamic invocation interface (DII).

Structure

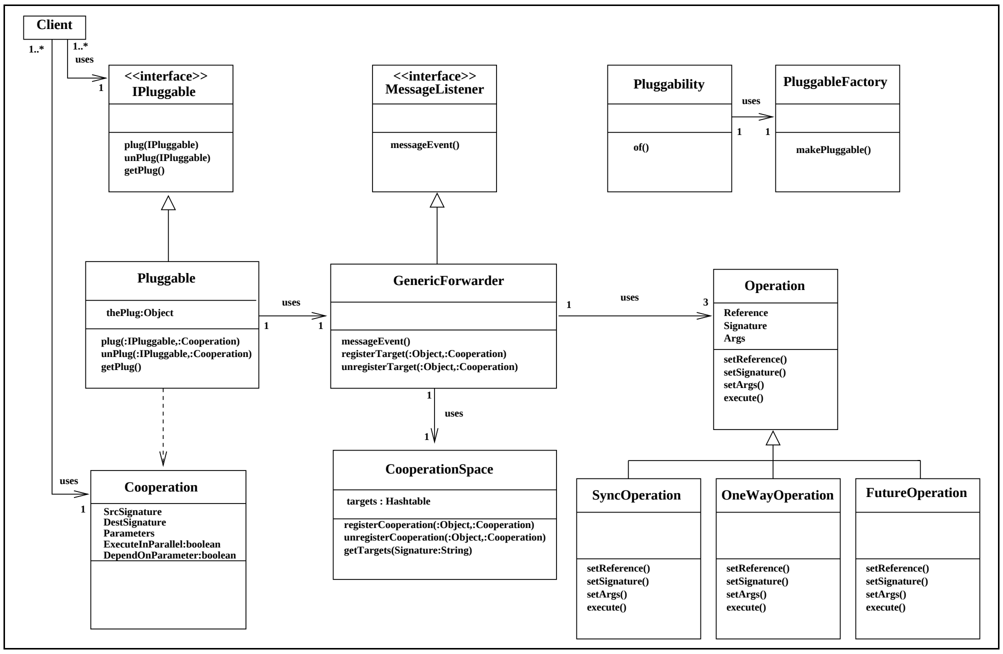

Figure 6: Plug-In Pattern Class Diagram

Participants

- IPluggable
  ∗ declares the interface that all classes have to implement in order to provide the plug-in service.
  ∗ provides methods for plugging in and out other plug-ins.
  ∗ provides a method for retrieving the object reference represented.

- Pluggable
  ∗ implements the IPluggable interface.
  ∗ holds a reference to an object (thePlug) it represents and adapts it to the IPluggable Interface (Object Adapter Pattern).
  ∗ delegates the message forwarding mechanism to a GenericForwarder.
  ∗ is responsible for the registration of the GenericForwarder as a MessageListener with the source object

- GenericForwarder
  ∗ implements a message listener interface
  ∗ dispatches message events, which are released by a supporting message listening service.
  ∗ defines methods for adding and removing target objects interested in a particular method call, specified by an object of class Cooperation.
  ∗ defines the behavior for interpreting the Cooperation objects when a method invocation in the source object is notified.
  ∗ forwards a method invocation in the source object to the target that is registered for this invocation by using a dynamic invocation mechanism.
  ∗ delegates the storage and target selection to an object of class CooperationSpace.

- CooperationSpace
  ∗ defines a method for storing Cooperation objects associated with object references, representing targets in a cooperation scenario.
  ∗ defines a method for retrieving a collection of target objects and their associated Cooperation objects
  ∗ implements the association of targets, Cooperation objects and method signatures

- Operation
  ∗ adapts a dynamic invocation mechanism.
  ∗ declares an abstract method for Operation execution (Command pattern).
  ∗ holds the context information for the dynamic invocation (target reference, method signature, parameters).
  ∗ defines methods for dynamically changing targets, methods and parameters of method calls.

- SyncOperation, OnewayOperation, FutureOperation
  ∗ implements different call semantics (synchronous, asynchronous, wait-by-necessity) based on the Operation class.

- Cooperation
  ∗ represents the context of a cooperaton scenario.
  ∗ stores metadata concerning dynamic method invocations in a implementation independent format.
  ∗ provides methods for storing and retrieving description of source and target method (signatures), a description of parameters that the target method expects and predicates that classify the parameter dependency (parallel, serial, parameter).

- PluggableFactory
  ∗ provides static factory methods in order to construct plug-ins at runtime.

- Pluggability
  ∗ provides a static method to add the plug-in capability dynamically to an object.

- Client (Agent)
  ∗ Client class that administers and manipulates plug-ins based on the IPluggable interface.

Collaborations

The Client is responsible for the correct plugging of the components. An object reference is made pluggable by passing the reference to the static `of()` method of the Pluggability Class and obtaining a reference to a Pluggable object (*see Figure 7*) The Client requests

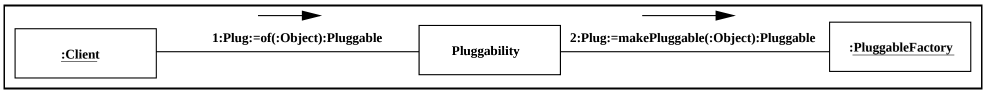

Figure 7: Pluggable Construction Collaboration Diagram

the metadata for the Cooperation objects from the components or retrieves them from a persistent repository. Holding two references to Pluggable objects, the Client passes the Destination plug to the `plug()` method of the source plug.

The object reference represented by the destination plug registers with the GenericForwarder for a message event. A message event is posted to the GenericForwarder in case of any method call issued to the object represented by the source plug.

The signature of the invoked method is used to dispatch the targets, which registered for this event. The Cooperation metadata is retrieved from the CooperationSpace for each target. A dynamic invocation is initialized with the directives and data from the Cooperation object and finally the invocation is performed (*see Figure 8*).

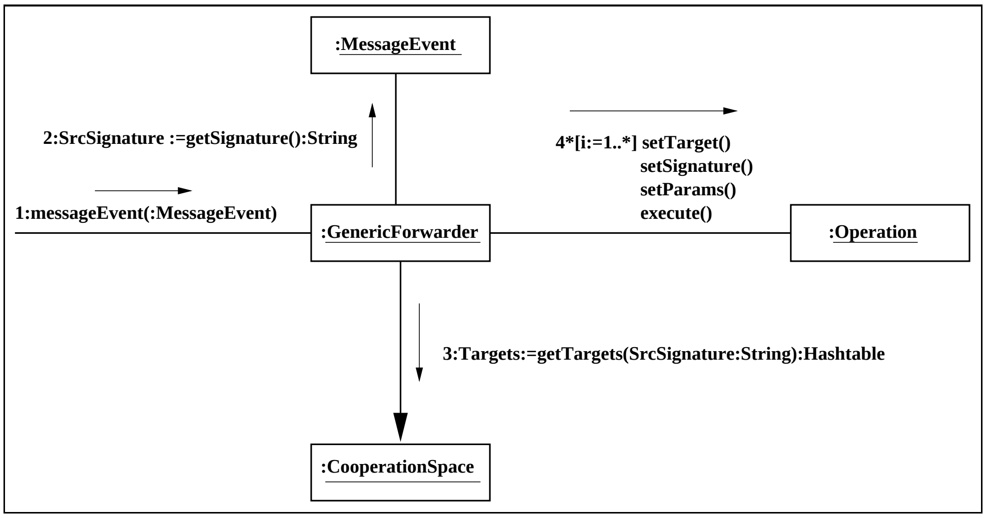

Figure 8: Message Forwarding Collaboration Diagram

Consequences

The Plug-In pattern

- facilitates cooperation between software components by establishing a unidirectional call forwarding mechanism from source plug to destination plug
- assigns the pluggability to any component at runtime
- abstracts a cooperation pattern, based on a signature and parameter description of the methods involved in a cooperation scenario
- decouples source component and target component by routing the cooperation through a message listening mechanism and a dynamic invocation of the target component

Implementation

As mentioned above, the implementation of the plug-in mechanism depends on the notification of incoming method calls sent to a component (provided through the so-called MessageListener interface), and on the dynamic invocation of methods (DII). Of course, these are not trivial to implement, but the focus lies on the abstraction of a cooperation between object-based components and the involved forces. We use ObjectSpace's Voyager for the support of these features. Actually, this means that all descriptions of method signatures are formatted for direct use by the Java Virtual Machine [Micb]. A concrete implementation considers the above use cases.

Notification of method invocation. The notification delivers a description of the called method, its signature and its parameters. This description can be used to query the cooperation space for the cooperating components.

Cooperation formation. The cooperation space can be implemented using hashtables for associating target references and their Cooperation objects with method descriptions as delivered by the method invocation notification. The Cooperation objects describe how to set up a call forwarding to each target component individually:

- which target method to call
- which parameters to forward
- if source and target method execute in parallel or in serial

The Cooperation space supports a multicast scenario. Depending on the cooperation description in the associated Cooperation object, the targets are grouped in two sets. The first group in due to execute in parallel, the second in serial with respect to the source method. The parameter selection and conversion mechanism is supported by a method in the Cooperation class. Of course, a parameter conversion can be arbitrarily complex. For parameters that have types related by inheritance the conversion is easy. More effort has to be spent when the types involved are completely unrelated. The knowledge needed for this conversion can be implemented in specialized subclasses of Cooperation.

Dynamic Invocation. For the parallel group, the call forwarding is done immediately when the call to the source method is issued. The serial group waits for the source method to terminate. The way this is done depends on the implementation of the message listening. Voyager casts a second event that indicates the end of the method execution. The call forwarding is done by the Operation objects, referenced by the GenericForwarder. Method signatures and parameters are passed together with the target references and the `execute()` method is called by the GenericForwarder. The different subclasses of Operation offer different possibilities to model dependencies of the target group involved in the cooperation.

Related Patterns

Adapter , especially the object adapter variant [GHJV95], is used to implement the dynamic assignment of the Pluggability to a component. Class Pluggable holds a reference to the object that is used as source or target in a cooperation scenario.

Facade Class Pluggable provides a Facade [GHJV95] to the plug-in pattern, consisting of a MessageListener, a GenericForwarder, a CoopearationSpace and an Operation object.

Command Class Operation implements an abstract Command [GHJV95] based on a dynamic invocation interface. The subclasses of Operation implement different call semantics. Class Operation stores all parameters needed for setting up a DII call. The call is performed by the `execute()` method.

4 Summary and outlook

In this paper, we have presented a framework to integrate negotiation capabilities, in particular negotiation strategies, into mobile agents dynamically. First, the framework's conceptual design, which is based on the concept of an *actor* system, was described. Then, a corresponding *plug-in* mechanism providing the technical basis to fulfill the requirements of dynamic composition was presented as a detailed design pattern.

The plug-in composition technique presented here does not match the requirements of the negotiation framework in an essential point yet. It does not provide a model for integrating rule dependent cooperation between components. The plug-in mechanism describes the what and when of an cooperation scenario. But *how* the cooperation between the coupled components should take place can depend upon events and conditions external to the cooperation itself. For instance, it can be desirable in a multi-issue/multi-lateral negotiation scenario that the sum of the single offers the agent makes to all his counterparts not exceed the total budget assigned to the agent. Such conditions can also emerge or be discovered at run-time, i.e. after the time of establishing a cooperation. In order to support dynamic condition checking, a flexible way to evaluate rules dynamically is currently being developed. This mechanism will be integrated into specialized subclasses of Cooperation, called ConditionalCooperation.

Within DynamiCS, a framework for dynamic rule management has been proposed [TGML99]. This framework contains a declarative representation of logical constraints modeled as Rule objects. These objects can be used as the representation of the conditions referenced in an cooperation scenario. Thus, condition evaluation is completely delegated to these objects. There are at least two points of integration possible. First is the pre-condition of a cooperation, second the post-condition of a cooperation. Only if these conditions evaluate to true, a cooperation is valid. This also allows for the support of distributed transactions performed by cooperating components. Another possibility is the integration of a *trigger list* into the Coordinator component. The trigger list represents a many-to-one relation between incoming message events and a target method. This is called an activation pattern. With an explicit representation of rules as components, a dynamic assembly of constraints is possible. They can be easily plugged into the framework. Establishing a cooperation with a rule object only means forwarding a call to the `evaluate()` method.

A different aspect which can also be influenced by rule management mechanisms is the explicit control of plug-ins' mobility. In the presented framework, this task is delegated to a coordination component monitoring the inner state of the agent (see 2.2). Since a DynamiCS agent can consist of an arbitrary number of plug-in components which are only loosely coupled (in the sense that no references of each other are held mutually) and therefore are not generally considered to be moved automatically together with the agent when it migrates to another place, a dedicated mobility management of an agent's components is necessary. But it is also obvious that a forced migration of *all* plug-ins belonging to an agent might not be the best solution in many cases. Another technical issue in managing agent mobility is that all active objects (*actors*) are running in their own thread of control. In Java, migration of threads is not a trivial task. One possible approach is to synchronize all activities inside a plugged component previous to migration. We are currently working on flexible rule-based strategies to solve these problems, which relate to agent management as well as to technical issues, adequately.

References

[Agh86] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Agh97] G.A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. Najm and J.-B. Stefani, editors, *Formal Methods for Open Object-based Distributed Systems.* Chapmann & Hall, 1997.

[Frø96] S. Frølund. *Coordinating Distributed Objects. An Actor-Based Approach to Synchronization.* MIT Press, 1996.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Westley, 1995.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[Ish98] T. Ishida, editor. *Community Computing – Collaboration over Global Information Networks.* Wiley, 1998.

[MD98] G. Meszaros and J. Doble. A pattern language for pattern writing. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design*, Software Patterns Series, chapter 29, pages 529–574. Addison-Wesley, 1998.

[Mey88] B. Meyer. *Object Oriented Software Construction*, page 23. Prentice Hall, 1988.

[Mica] SUN Microsystems. Java. http://java.sun.com.

[Micb] Sun Microsystems. The java virtual machine specification. http://java.sun.com/docs/books/vmspec/index.html.

[MZD98] A.-I. Mouaddib, S. Zilberstein, and V. Danilchenko. New directions in modeling and control of progressive processing. In Henri Prade, editor, *ECAI 98. 13th European Conference on Artificial Intelligence*. John Wiley & Sons, Ltd., 1998.

[Obj] ObjectSpace. Voyager. http://www.objectsapce.com/Voyager.

[Oli96] Jim R. Oliver. *On Artificial Agents for Negotiation in Electronic Commerce*. PhD thesis, The Wharton School, University of Pennsylvania, 1996.

[RMdS98] A. Romao and M. Mira da Silva. An agent based secure internet payment system for mobile computing. In W. Lamersdorf and M. Merz, editors, *Trends in Distributed Systems for Electronic Commerce, Proceedings of the International IFIP/GI Working Conference TREC 98*, Lecture Notes in Computer Science, pages 80–93, Hamburg, Germany, June 1998. Springer, Berlin Heidelberg New York.

[TGML98] M.T. Tu, F. Griffel, M. Merz, and W. Lamersdorf. A plug-in architecture providing dynamic negotiation capabilities for mobile agents. In K. Rothermel and F. Hohl, editors, *Proc. 2. Intl. Workshop on Mobile Agents, MA'98, Stuttgart*. Springer LNCS, 1998.

[TGML99] M.T. Tu, F. Griffel, M. Merz, and W. Lamersdorf. Interaction-oriented rule management for mobile agent applications. In *Proc. of the Second Int. Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*. Kluwer Academic Publisher, June 1999.

[TLGL99] M.T. Tu, C. Langmann, F. Griffel, and W. Lamersdorf. Dynamische Generierung von Protokollen zur Steuerung automatisierter Verhandlungen. In *Proc. 29. Jahrestagung der Gesellschaft für Informatik (Informatik'99)*. Springer LNCS, 1999. (In German).

[VMK98] H. Vogler, M.-L. Moschgat, and T. Kunkelmann. Enhancing mobile agents with electronic commerce capabilities. In M. Klusch and G. Weiss, editors, *Cooperative Information Agents II, Proceedings of the 2nd International Workshop CIA 98*, Lecture Notes in Artificial Intelligence, pages 148 – 159, Paris, France, July 1998. Springer, Berlin Heidelberg New York.

[Zil96] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.