AN ACTOR-BASED FRAMEWORK FOR NEGOTIATING MOBILE AGENTS

M.T. TU, C. SEEBODE, F. GRIFFEL AND W. LAMERSDORF *

Distributed Systems Group, Computer Science Department
University of Hamburg, Germany
Vogt–Kölln–Str. 30, 22527 Hamburg, Germany
[tu,1seebode,griffel,lamersd]@informatik.uni-hamburg.de

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" [1], 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. 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 [2].

This paper presents a component architecture of self-interested negotiating mobile agents. It is implemented in the context of the DynamiCS (*Dynamic*ally *C*onfigurable *S*oftware) 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 ability to negotiate autonomously – e.g. to bid at Internet auctions – can be considered a useful intelligent capability which is directed to a clear goal, i.e. finding the best possible deal according to a given value function. However, even when restricted to E-Commerce scenarios, the term negotiation can still cover many different types of processes, during which the participants try to achieve some kind of *common agreement*. Therefore, we proposed

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

a generic *protocol* specification language to precisely express the semantics of a negotiation type [2,3]. Moreover, 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. In other words, 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 discussed in the publications cited, this paper will have a specific focus on how to embed different negotiation strategies into mobile agents and the corresponding plug-in mechanism.

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 was chosen as the implementation language and Voyager [4] 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.

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. 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.

2 An actor-based negotiation framework

The development of self-interested negotiating agents requires an understanding of the sequence of events in a negotiation. The structuring of the overall task of a negotiating agent into specialized *modules* which can be dynamically plugged into a mobile agent (or agent frame) 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 (see [2] for a classification of negotiation strategies). 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 [7]. 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 are modeled as special actors that control message dispatch to a group of actors (see Figure 1). Coordinators are responsible for constraining the execution of their controlled actors by defining a set of rules that have to be evaluated before dispatching messages to a group of actors. This is provided by Collaboration classes that are part of the plug-in mechanism (see Section 3). The Collaboration classes implement methods and constraints that collaborate in an actor group. Conceptually, they perform part of the functionality of *Synchronizers* [8].

With coordinators and actors as roles in the execution model, a *hierarchical*

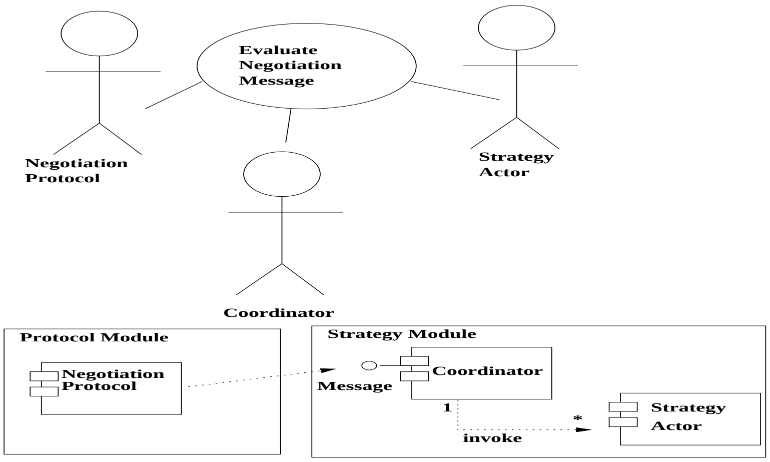

Figure 1. Roles and their distribution in the negotiation framework

composition of strategies is possible (see Figure 3). 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 [9]) 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.

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 [6] 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).

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 ac-

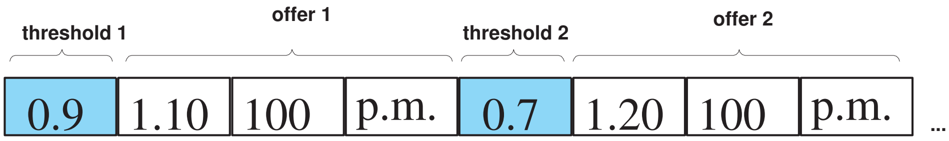

Figure 2. Numerical example of an evolution-based strategy

tor which returns the next element of the offer sequence in each atomic instruction. However, it can easily be seen that the performance of the overall genetic algorithm can be improved by using several actors concurrently, 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.

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 3 illustrates this technique for two levels.

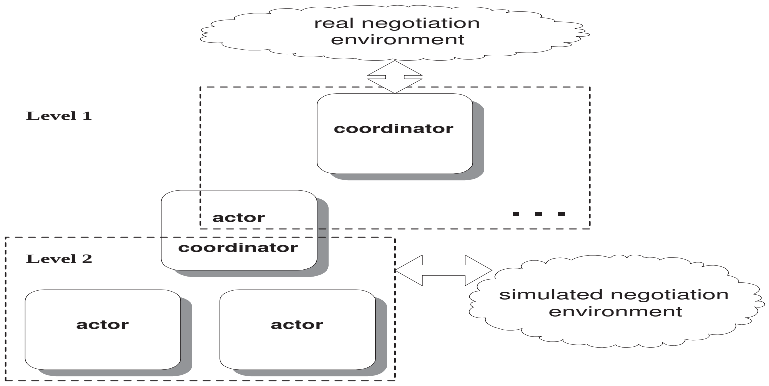

Figure 3. Structure of a hierachical strategy with 2 coordinator-actor levels

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), recognizable quality (meaning the quality of an approximate result can determined at run time) and monotonicity (meaning the quality of the result is a nondecreasing function of time and input quality).

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 Basic conception of the plug-in mechanism

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 [10]:

- *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.

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 from 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. 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. At runtime, however, one component can play different roles in different cooperation scenarios. Hence, on the technical level, 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 (see also [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 plug-in mechanism can be decomposed into three complementary actions: notification on message invocation, cooperation formation and dynamic invocation. In order to enforce the runtime relation between method invocations in cooperating components, the implementation of the plug-in mechanism 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 (see also [2]), these services are provided by ObjectSpace's Voyager Framework [4]. The technical design of the message forwarding mechanism is illustrated by Figure 4.

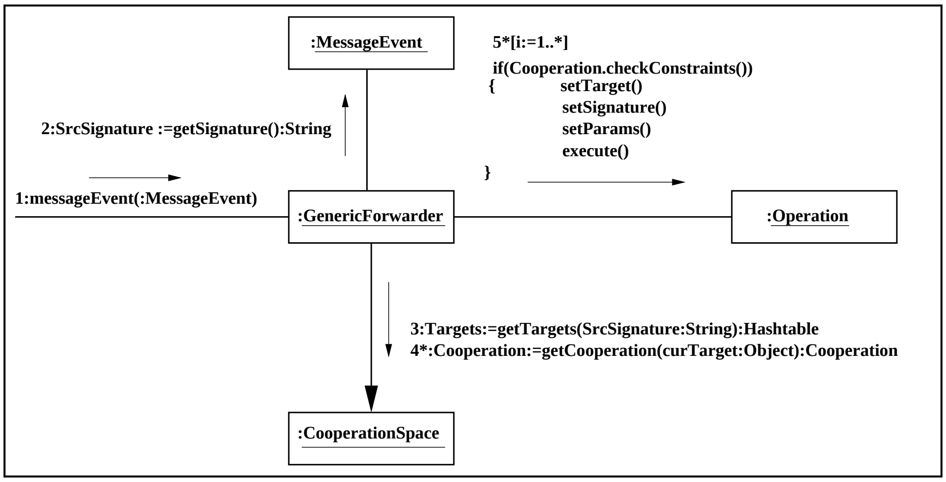

Figure 4. Message Forwarding Collaboration Diagram

3.2 Applying the plug-in mechanism to actor-based strategies

The actors constituting the strategy module are just another example of object-based components which can be asssembled by means of the plug-in mechanism. An actor collection represents a parallel system, every actor that is instantiated inside the module works concurrently to the other actors. The complexity of designing a parallel actor system can be compared to the complexity of designing parallel algorithms. In order to design actor-based strategies, one has to decide

- which tasks run in parallel.
- which synchronization constraint apply between the parallel tasks.

The synchronization constraints for the actor-based strategies are conceptually represented by the strategy constraints (see Section 2.2). The tasks are mapped to actors, whereas the synchronicity is achieved by the plug-in mechanism. Actors operate asynchronously. The primitive operation that implements this asynchronous behavior is the `send()` method. The results of any computation done by the actors are communicated asynchronously as well.

As an example let's assume the genetic algorithm mentioned above. The purpose of this algorithm is to improve strategy quality over time. The data structure the GA operates on is the strategy or a collection of strategies. The intuitive approach of mapping different GAs to different actors can be implemented by providing different fitness functions in order to perform different computations inside the GA. For the sake of simplicity, let's assume an agent starts the GA every time a new offer arrives. The task to be performed by one actor consists of (randomly) selecting a population, running / testing these strategies and generating a new population of strategies by GA. The iteration of these instructions should provide better strategies with each cycle. The collection of actors performing the GA is controlled by the coordinator (which is itself an actor) which

- creates GA actors
- controls invocation of actor cycles
- collects results

The invocation of a new computational cycle is achieved by each actor by sending an invocation message to itself. The control of this invocation is done by the coordinator through the plug-in mechanism. The coordinator is plugged into the GA actors. This means the coordinator registers for the invocation messages with the GA actors. These message invocations are forwarded to the coordinator. The forwarding depends on the evaluation of synchronization constraints held by the coordinator. These constraints concern computation time or result qualilty. The conditional dispatch of these constraints can lead to the stop of computation or requesting the current result from the GA actor or also the creation of new actors to improve result quality. The framework provides *adapter* classes for the integration of such actor implementations. Currently the Actor Foundry package [5] is being integrated.

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 and cooperation between actors was presented.

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.

References

1. T. Ishida, editor. *Community Computing – Collaboration over Global Information Networks.* Wiley, 1998.
2. 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.
3. 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).
4. ObjectSpace. Voyager. http://www.objectsapce.com/Voyager.
5. T.H. Clausen. The actor foundry. http://www-osl.cs.uiuc.edu/foundry/index.html.
6. Jim R. Oliver. *On Artificial Agents for Negotiation in Electronic Commerce.* PhD thesis, The Wharton School, University of Pennsylvania, 1996.
7. G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
8. S. Frølund. *Coordinating Distributed Objects. An Actor-Based Approach to Synchronization.* MIT Press, 1996.
9. S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.
10. G. 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.