DynamiCS: 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
E-mail: [tu,1seebode,griffel,lamersd]@informatik.uni-hamburg.de

In this article, 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. Additionally, this contribution describes how interaction-oriented rule mechanisms can be deployed to control the behavior of strategy actors.
Keywords: negotiation, mobile agent, interaction patterns, rule-sensitive actors, 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

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

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 (Dynamically Configurable Software) 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 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 contribution will have a specific focus on how to embed different negotiation strategies into mobile agents by using a dynamic *plug-in* mechanism and how to control the behavior of actors and strategies by using interaction-oriented rule mechanisms. (This article is an extended version of the contribution to the IAT'99 Workshop on Agents in Electronic Commerce (WAEC'99) [14] with a new section on rule-sensitive actors.)

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 describes how a decentralized, interaction-oriented rule mechanism can be deployed to control the behavior of actors implementing a strategy. Section 5 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.

2.1. Negotiation strategies

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). Negotiation in general can be considered under several aspects. It has been described as a process contributing to conflict resolution [16], task allocation [17] and resource allocation [18]. All these aspects contain an element of behavioral control of the agent. This control is part of the negotiation strategy and administers the effort that the agent is going to spend in order to achieve a desired outcome. There is always a tradeoff between the best possible action an agent can compute and the resources that are used.

Building a framework for the development of negotiation strategies means to explicitly model the requirement of resource control. It will be used to model the intelligence of the agent and assures responsiveness, liveness and result quality. A particular strategy models the intelligence by using specialized data structures and knowledge of the problem domain. A generic framework for the development of negotiation strategies 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.

In consequence, the framework contains all the logic to support an *execution* model of the strategy. 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 possible actions taken by a negotiating agent is a challenging task for the design of an execution model. On the one hand, such 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, the agent plays a reactive role when responding to messages generated by other agents.

2.2. The Framework model

The execution model of a particular strategy is supported by a number of active objects. Active objects correspond to the notion of an ACTOR system [7]. An *actor* executes in response to message passing and performs atomic in-

structions that are executed 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*.

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 Coordination classes that are part of the plug-in mechanism (see Section 3). Furthermore, the Collaboration classes implement methods and constraints that collaborate within an actor group. Conceptually, they perform part of the functionality of *Synchronizers* [8].

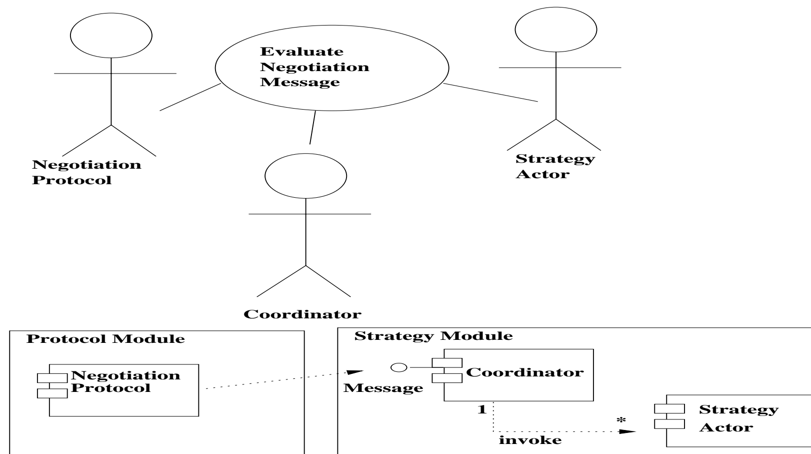

Figure 1. Roles and their distribution in the negotiation framework

The simplest possibility to implement a strategy based on the execution model of this framework is to combine one coordinator, one actor and one synchronizer. A hierarchical composition of these simple execution units is possible (see Figure 3). Such a composition could represent different stages of computation and result quality as in a combination of different *progressive processing units* [12].

Another possible example of mapping actors to 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.

2.3. 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).

All these constraints can be expressed by introducing coordination synchronizers that control the message dispatch to the actors. 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. Conceptually,

this kind of control can be performed by specifying timing constraints between the actors [19].

2.4. 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, an approach of integrating genetic algorithms into the strategy framework is presented. 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).

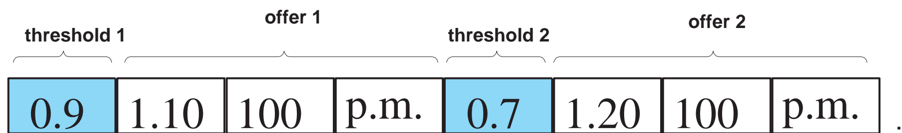

Figure 2. Numerical example of an evolution-based strategy (following [6])

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. 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 hierarchy 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.

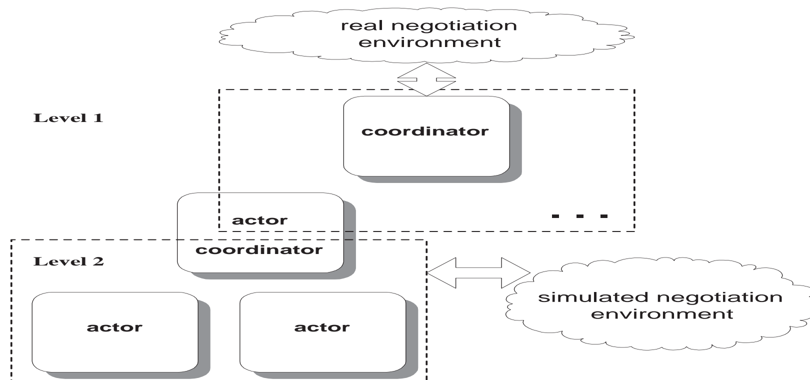

Figure 3. Structure of a hierarchical strategy with 2 coordinator-actor levels

Also, with respect to the anytime properties mentioned above, it can easily be shown that such a hierarchical 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 introduced 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 [11]:

- *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 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 4.

`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

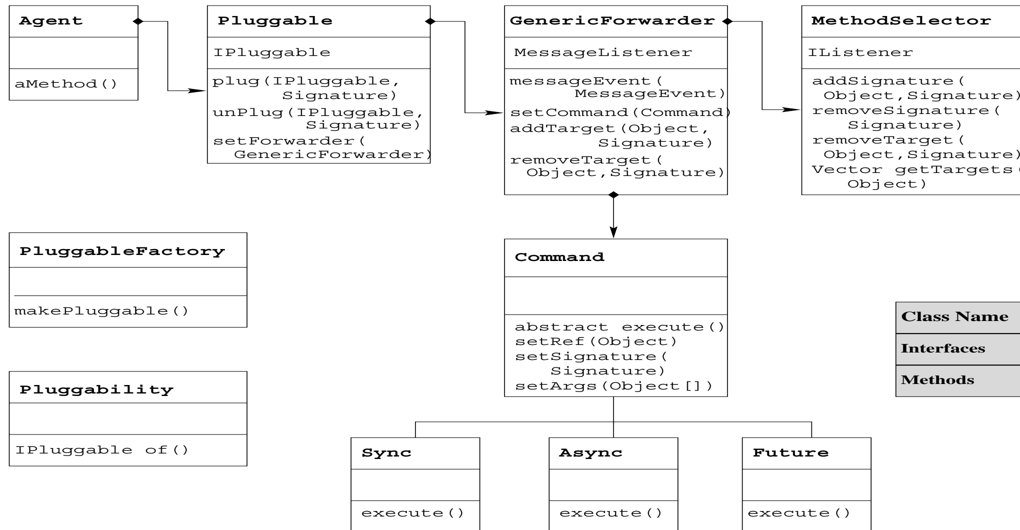

Figure 4. Class diagram for plug-in mechanism

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).

The usage of the `Pluggability` class is illustrated in (see Figure 5). The Client (i.e. caller of the class methods) 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.

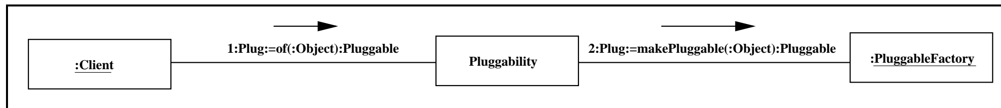

Figure 5. Pluggable Construction Collaboration Diagram

The Client requests the metadata for the Cooperation objects from the components or retrieves them from a persistent repository. After that, 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 any method call is made to the object represented by the source plug.

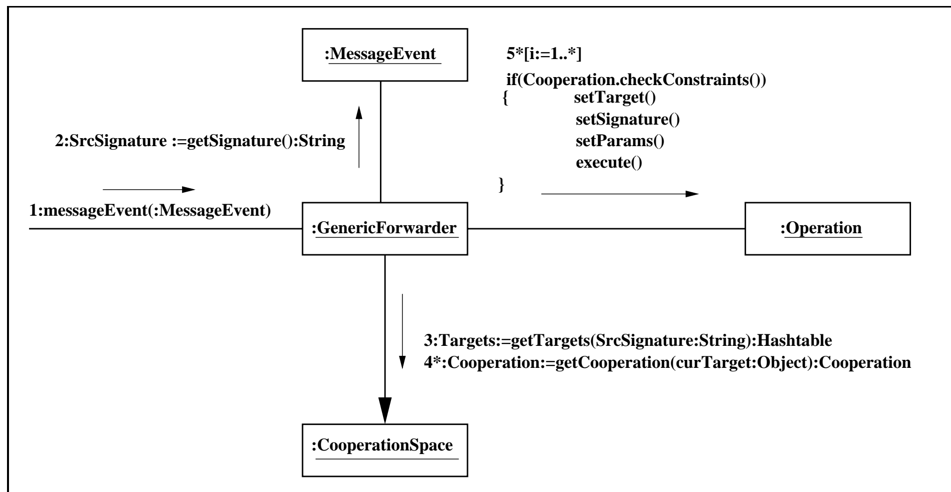

Figure 6. Message Forwarding Collaboration Diagram

The technical design of the message forwarding mechanism is illustrated by Figure 6. The signature of the invoked method is used to dispatch the method call

to all targets which registered for this event. The `GenericForwarder` makes use of `Cooperation` metadata retrieved from the `CooperationSpace` for each target to map the called signature including parameters to the corresponding signature of the target (which may differ from the signature used by the caller). A dynamic invocation is initialized with the directives and data from the `Cooperation` object and then the invocation is performed.

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 2.3). 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 quality. 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. Using rule-sensitive actors to control strategies

One of the most important questions regarding automated strategies for negotiations is how to control their behavior, especially when some kind of dynamic and self-adapting strategies such as the genetic algorithms mentioned above is deployed[1]. Using a *hierarchical* architecture consisting of coordinators and subactors as proposed in Section 2.2, a principal answer to this question is already given: The coordinator is responsible for collecting and evaluating the results provided by his subactors, thus having the possibility of eliminating results that violate certain *control criteria*. However, this answer is still too abstract since it does not state how the coordinator can formulate control criteria and enforce them on the subactors. Moreover, it would be much more effective if the subactors themselves could be controlled in such a way that rule violating results cannot be generated or provided to a higher level in the hierarchy. In this section, we show how a generic rule management mechanism can be used to control the behavior of actors or the corresponding strategies, respectively.

[1] It seems obvious that in open and constantly changing market environments such as the booming internet auctions, static strategies would not be appropriate.

The rule system introduced here has been described in detail in other publications, most notably [13], and supports the following requirements:

- Interaction support: Rules issued from one party can be enforced as requirements on *another* party. Moreover, rules of different parties can be matched with each other to provide a common basis the pending interaction. Section 4.1 explains how this feature is supported.
- Decentralization: In order to meet the requirements of mobile agents or software components in general as autonomous, encapsulated entities, rules should not be managed centrally, but rather held directly by the applications which serve as *rule containers*.
- Object- and Event-Orientation: Although rules are semantically add-ons for applications, they should be implemented as first-class objects which can activate themselves upon occurences of corresponding events, since this enables an efficient decentralized rule management. Every rule object directly acts as an event listener that only reacts to events of types belonging to its trigger list.
- Mobility: Since rules should be (technically) first-class objects that can be added to mobile agents at run-time, it would be efficient (and also more elegant) to have rules themselves implemented as mobile objects that can migrate from one agent to another.
- Transactional activation: In case the activation of a requirement rule or a policy which happens after some activity, i.e. one or more function calls, has been carried out in the application, results in *false*, then it should be possible to rollback the respective activity so that the violation of rules can be avoided to a maximal degree. This implies that rule-sensitive activities have to be carried out as transactions.

4.1. Interaction support

The rule system provides support for interactive behavior on two levels: On the *logical* one, a *unification* function is provided to calculate the common basis of two rule expressions which is defined as the weakest expression that implies both inputs (see [13] for detail). The following example illustrates this function.

Example 1. Given
P1 = ((cost $\leq$ 50) $\wedge$ (speed $\geq$ 5)) $\vee$ ((cost $\leq$ 100) $\wedge$ (speed $\geq$ 10))
P2 = ((cost $\leq$ 90) $\wedge$ (speed $\geq$ 20)) $\vee$ (speed $<$ 5)
P1' = (cost - 2 * speed $\geq$ 0)
P2' = (cost - 2 * speed $\leq$ 0)
Then
unify(P1, P2) = (cost $\leq$ 90) $\wedge$ (speed $\geq$ 20)
unify(P1', P2') = (cost - 2 * speed = 0)

Secondly, on the *activation* level, different *modes* to activate rules are provided. For example, in a warehouse scenario, a customer should be able to pass his rule with respect to payment modalities as a requirement to be fulfilled by the provider, i.e. to be activated on the warehouse side. Therefore, the following activation modes are proposed to support such a remote activation concept besides the local activation.

In general, the activation of a rule can take place in two modes, namely INTERNAL or EXTERNAL. In the former case, the activation semantics of the corresponding rule type is applied to the component the rule belongs to. In the latter case, the rule is applied to an external component, the reference of which is passed as part of the triggering event. To enable different kinds of interaction semantics, the EXTERNAL mode is subdivided into the following variants:

- ONEWAY: In this mode, a copy of the rule semantics (i.e. condition, activation and mode) is created and passed to the external component for activation. (Thereafter, the copy is deleted.)
- CALLBACK: In addition to the ONEWAY semantics, the external component returns the allocation of the properties which are referred to in the rule as the *common cooperation basis* so that both sides can enforce the exactly same configuration with respect to these properties.
- FILTER: In order to maintain the autonomy of application components, the activation of external rules should not take place directly, but only through the use of corresponding *filter rules.* That means, when an external rule of mode ONEWAY or CALLBACK is activated, there must exist a corresponding rule of mode FILTER kept at the target component, with which the first one is unified. The result of the unification, if not empty, is then activated.

4.2. Rule activation

In order to enable the integration of rule functionality into the application level in a *transparent* way, a rule-sensitive dynamic invocation interface (RS-DII) is provided which can be used by any application component to carry out method calls (local or remote) in a rule-safe way, using the usual API very similar to the DII of CORBA products. However, the RS-DII offers two additional features:

- When a method is called, two instances of `MethodRuleEvent` are generated, one *before* and one *after* the call is performed using an usual DII mechanism. During the method call, any rule exceptions[2] thrown by rule objects are caught and forwarded to the application object (making the call) and the call is interrupted.
- A *distributed transaction service* is employed to roll-back the method in case a rule exception has been thrown *after* the method has been carried out.

Two versions of the RS-DII have been developed, the first of which – called `Dynamic` – is used to make a single call, and the second – called `DynamicVector` – is used to perform several calls in a rule-sensitive and transactional manner. The following figure illustrates the enforcement of a coordinator's rule on a subactor by using the CALLBACK mode.

`coordinator` and `actor` represent two rule-sensitive actors supporting the `RuleContainer` interface. Now, when the `coordinator` calls a method offered by the `actor` using the RS-DII (1), this mechanism first delays the call and pushes a `MethodRuleEvent` into the event channel which delivers it to all rule objects (implementing the `RuleEventListener` interface) which are listening for the corresponding event type and contextID (2). In this case, a rule of the `coordinator` in ONEWAY mode is triggered and due to the semantics of this mode, this rule first creates a copy of itself, packs it as a `unifyParam` into a `FilterRuleEvent` and sends this event to the event channel (3), thereupon the corresponding FILTER rule of `actor` is triggered. There, both rules are matched using the `unify` method, and if the result is non-empty, it is activated on `actor`'s side. This ensures that certain parameters for the negotiation action – e.g. the allowed budget for making offers – are set correctly. Then, a `ReplyRuleEvent` is sent back to the ONEWAY rule of the `coordinator` signaling that the rule copy

[2] Note that rule (violation) exceptions are thrown by means of events of type `ReplyRuleEvent` (s. [13]).

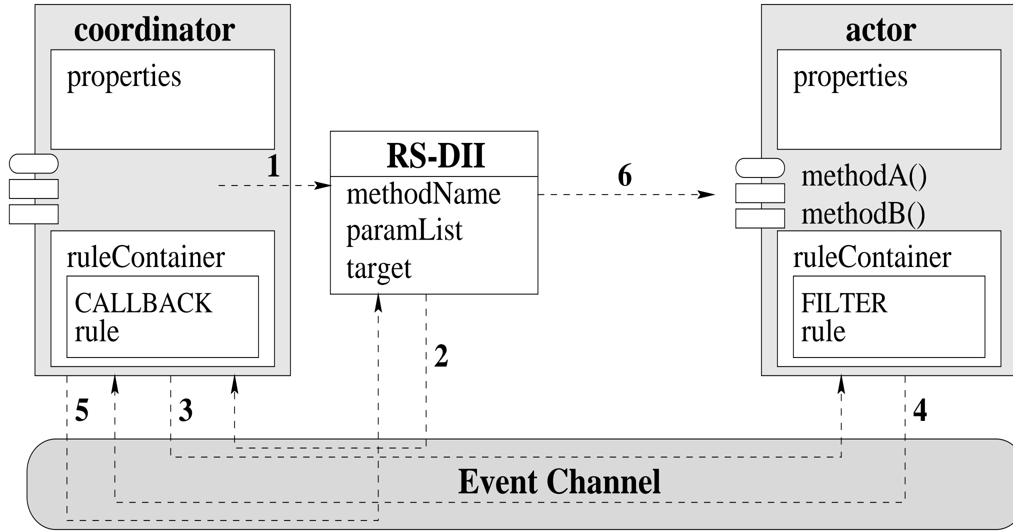

Figure 7. Invocation of a rule-sensitive call

has been activated successfully at the remote site (4). Upon this, the originating rule issues another `ReplyRuleEvent` to the RS-DII component meaning that the rule has been activated successfully (5) and the method call can now be carried out in the usual way (6). However, *after* the method has been performed by `actor`, a similar chain of events is generated and in case any rule exception is thrown, the RS-DII component will enforce the process to roll-back the method. In this way, it is garanteed that certain negotiation constraints are not violated if when the method call has completed.

5. 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. After that, it was shown how a generic rule management mechanism can be activated between coordinators and subactors to garantee that certain constraints are not violated before and after the interaction which is carried out as a transaction.

A different aspect which can also be influenced by rule management mech-

anisms 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 2.3). 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.

Another important aspect of actors implementing a strategy which cannot be solved by external rule mechanisms alone is explicit control of computing resources, especially computation time. Although the genetic algorithms we have proposed (and already implemented within the DynamiCS project) are principally appropriate for resource-bounded computation, implementing a *generic* framework to integrate the any-time features mentioned above into strategy actors is not a simple task which we are currently investigating. [10] provides a good description of this kind of computation control.

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] 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.

[11] 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.

[12] S. Zilberstein and A. Mouaddib. Reactive Control of Dynamic Progressive Processing. In *Proc. of the 16th International Joint Conference on Artificial Intelligence. Stockholm, Sweden*, August 1999.

[13] 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.

[14] M.T. Tu, C. Seebode, F. Griffel and W. Lamersdorf. An actor-based framework for mobile negotiating agents. In *Proc. IAT99 Intl. Workshop on Agents in Electronic Commerce (WAEC'99), Hongkong*, 1999.

[15] G. Zlotkin and J.S. Rosenschein. Negotiation and conflict resolution in non-cooperative domains. In *Proc. of the Eighth natinal Conference on Artificial Intelligence, Boston, Mass.*, July 1990.

[16] S.E. Lander and V.R. Lesser. Customizing distributed search among agents with heterogenous knowledge. In *Proc. of the First International Conference on Information and Knowledge Management, Baltimore, Maryland*, pages 335–344, November 1992.

[17] E.H. Durfee and Thomas A. Montgomery. A hierarchical protocol for coordinating multiagent behaviours. In *Proc. of the Eighth National Conference on Artificial Intelligence, Boston, Mass.*, pages 86–93, August 1990.

[18] S.E. Conry, K. Kuwabara, V.R. Lesser and R.A. Meyer. Multistage negotiation in distributed constraint satisfaction. In *IEEE Transactions on Systems, Man and Cybernetics. Special Issue on Distributed Artificial Intelligence*, January 1992.

[19] S.Ren and G.A. Agha. RTSynchronizer: Language support for Real-Time Specifications in distributed systems. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, La Jolla, California*, June 1995.